VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

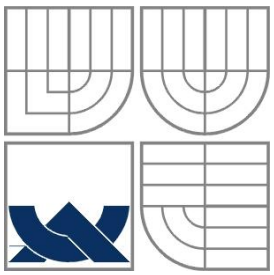
OPTIMALIZACE STRUKTUROVANÝCH DOTAZŮ NAD ROZSÁHLÝMI DATABÁZEMI

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

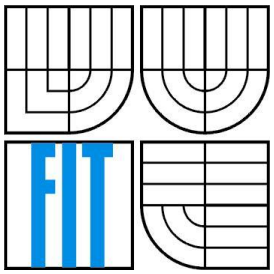
AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ JANEČEK

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE STRUKTUROVANÝCH DOTAZŮ NAD ROZSÁHLÝMI DATABÁZEMI

OPTIMIZATION OF STRUCTURED QUERIES ON LARGE DATABASES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ JANEČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Radek Burget Ph.D.

BRNO 2012

Abstrakt

Tato diplomová práce se zabývá optimalizací strukturovaných dotazů nad rozsáhlými databázemi. Principy těchto optimalizací jsou později využity při tvorbě aplikace, která umožňuje vyhledávání nad jednou konkrétní rozsáhlou databází. Současně tato práce porovnává efektivitu nově navržených konstrukcí SQL v porovnání s konstrukcemi neoptimalizovanými.

Abstract

This master's thesis deals with the optimization of structured queries on large databases. The principles of the optimizations are used during the creation of an application, which allows making queries over a specific large database. At the same time, this thesis compares the efficiency of the new designed SQL constructions with the non-optimized SQL constructions.

Klíčová slova

dotaz SQL, optimalizace dotazů SQL, databáze MySQL, kaskádové zpracování, prováděcí scénář, skupiny atributů

Keywords

SQL query, optimization of SQL queries, MySQL database, cascade processing, process scenario, attribute groups

Citace

Janeček Jiří: Optimalizace strukturovaných dotazů nad rozsáhlými databázemi, diplomová práce, Brno, FIT VUT v Brně, 2012

Optimalizace strukturovaných dotazů nad rozsáhlými databázemi

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Radka Burgeta Ph.D.
Další informace mi poskytli Pavel Šimon, Pavel Krčma z AVG Technologies s.r.o.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Janeček
29. 2.2012

Poděkování

Za vedení při tvorbě této zprávy bych chtěl poděkovat doktoru Burgetovi a kolegům z AVG Technologies s.r.o., kteří mi byli oporou při tvorbě softwarové části.
Současně bych chtěl poděkovat Ing. Kunetkovi za odhodlání a odvahu při korektuře této diplomové práce.

Obsah

Obsah ... 1
1 Úvod ... 3
2 Analýza prostředí ... 4
2.1 Struktura databáze virdat ... 5
2.2 Implementace databáze virdat ... 10
3 Možnosti optimalizace dotazů nad rozsáhlými databázemi ... 16
3.1 Optimalizace SQL dotazů pro databáze MySQL ... 16
3.1.1 Optimalizace přístupu k datům ... 16
3.1.2 Restrukturalizace dotazu ... 17
3.1.3 Současné schopnosti optimalizátoru MySQL ... 19
3.1.4 Optimalizace specifických typů dotazů ... 20
3.1.5 Pokyny pro optimalizátor MySQL ... 22
3.2 Testování výkonu dotazu ... 24
3.2.1 EXPLAIN ... 25
3.2.2 Čas ... 25
3.2.3 QPM ... 26
3.3 Program QueryCounter ... 27
3.3.1 Implementované testovací metody ... 27
3.3.2 Možnosti vstupů ... 27
3.3.3 Formát textového souboru ... 28
3.3.4 Implementace a technické detaily programu QueryCounter ... 30
4 Návrh systému AVGMIS ... 32
4.1 Dotazy – jádro aplikační logiky ... 32
4.1.1 Požadované atributy a jejich možné četnosti ... 32
4.1.2 Řízení vyhledávání – tabulka ctrl_avgmis ... 37
4.1.3 Algoritmus řídící vyhledávání ... 38
4.1.4 Kaskádové zpracování a hierarchie scénářů ... 47
4.1.5 Zkrácené vyhodnocení a omezení velikosti výsledku ... 51
4.1.6 Struktura navrhovaných dotazů SQL ... 53
4.1.7 Typy dočasných úložišť ... 59
4.2 Grafické uživatelské rozhraní ... 60
4.2.1 Aplikace VTMIS – předloha aplikace AVGMIS ... 60
4.2.2 Styl navrhovaného grafického uživatelského rozhraní ... 61
4.2.3 Vstup ... 62

4.2.4 Výstup 64
5 Implementace systému AVGMIS 67
5.1 Programová struktura systému AVGMIS 67
5.1.1 Parser 68
5.1.2 Assembler 69
5.1.3 Loger 70
6 Testování systému v reálném nasazení 71
6.1 AVGMIS – analytické rozhraní 72
6.1.1 `log_avgmis` – úložiště záznamů 72
6.1.2 Vstup 73
6.1.3 Výstup 73
6.2 Hodnocení využití systému AVGMIS během testovacího provozu 74
7 Hodnocení optimalizací 77
7.1 Obecné optimalizace 77
7.2 Optimalizace použité v aplikaci AVGMIS 80
7.2.1 Skupiny strukturálních indexů a `files` 81
7.2.2 Skupina detekcí 84
8 Možná další vylepšení 86
9 Závěr 88
10 Literatura 89
11 Příloha A 90
12 Příloha B 94
13 Příloha C 98
14 Příloha D 113

1 Úvod

Hledání v databázích. Činnost, která je v dnešní době zcela samozřejmá. Miliony lidí na celém světě tuto činnost provozují. Nezávisle na informacích, které hledají, mají všichni tito lidé bezesporu dva společné cíle. Prvním cílem je najít hledanou informaci. To obvykle závisí na tom, jestli hledají na správném místě, a nutno dodat, že s tímto cílem jim tato práce nepomůže. Druhým, a pro tuto diplomovou práci zásadním cílem, je snaha najít hledanou informaci či fakt o její neexistenci co možná nejrychleji.

K tomu, aby byla informace nalezena v co možná nejkratším možném čase, jsou databázové stroje vybaveny nejrůznějšími mechanismy. Od optimalizátorů dotazů přes indexy, které obtěžkávají tabulky naplněné informacemi, až po různé formáty úložišť (angl. storage engines) jednotlivých tabulek, které budou reflektovat naši snahu o co možná nejrychlejší získání informace.

A právě zde se nabízí otázka, jestli všechny tyto mechanismy jsou dostatečnou zárukou, že hledaná informace se k uživateli dostane v rámci možností co nejrychleji. Jistě by bylo velmi pohodlné si odpovědět „ano“ a nechat všechnu tu práci na databázovém stroji. Jen ať provádí optimalizace našich dotazů, které mohou být faktické správné, ale nemusí být ani zdaleka efektivní.

Tato práce si klade za cíl čtenáři vysvětlit některé techniky tvorby efektivních dotazů a posléze ukázat, jak se tyto techniky dají v praxi aplikovat na konkrétní, již existující databázi. Nepůjde zde jen o pouhé přetvoření databázových dotazů pomocí jiných konstrukcí jazyka SQL, které dovedou najít stejnou množinu dat, ale budou zde diskutovány i techniky, které umožní urychlit nalezení hledané množiny dat.

Již teď na úvod si můžeme říct, že podstatou této práce je převzetí části úkolů optimalizátoru databázového stroje. O využitelnosti a potřebě těchto optimalizací se můžete přesvědčit buď v těle práce, nebo v jejích přílohách, kde jsou uvedeny výsledky prováděných optimalizací.

Tato práce se zabývá teoretickým rozborem možných optimalizací pro konkrétní databázový stroj (MySQL). Dále se pak práce zabývá návrhem jednotlivých databázových dotazů pro jednu konkrétní databázi. V další části práce je proveden návrh aplikace, která by měla využívat těchto nově navržených dotazů. Předposlední část práce se zabývá implementací této aplikace. A konečně poslední část pak hodnotí dosažené výsledky optimalizovaných dotazů. První tři části (rozbor dotazů, návrh dotazů, návrh aplikace) jsou obsaženy v rámci semestrálního projektu.

2 Analýza prostředí

V předchozí kapitole jsem uvedl, že všechny optimalizace budou prováděny nad existující databází. To v důsledku znamená, že schéma databáze je neměnné, a proto se zde nebudu zabývat tvorbou struktury databáze, normalizací jejích tabulek či případnou další optimalizací vytvořených konstrukcí DDL.

V následujících odstavcích této kapitoly bude popsána struktura databáze `virdat` a její implementace. Současně, vzhledem k velikosti a členitosti databáze, se omezím pouze na atributy (tabulky, sloupce, …), které budou mít význam pro později navrhovanou aplikaci.

Než se pustím do popisu struktury databáze, uvedu zde několik informací o databázovém stroji, aby čtenář získal představu o velikosti databáze a o hardwaru, na kterém je tato databáze spuštěna:

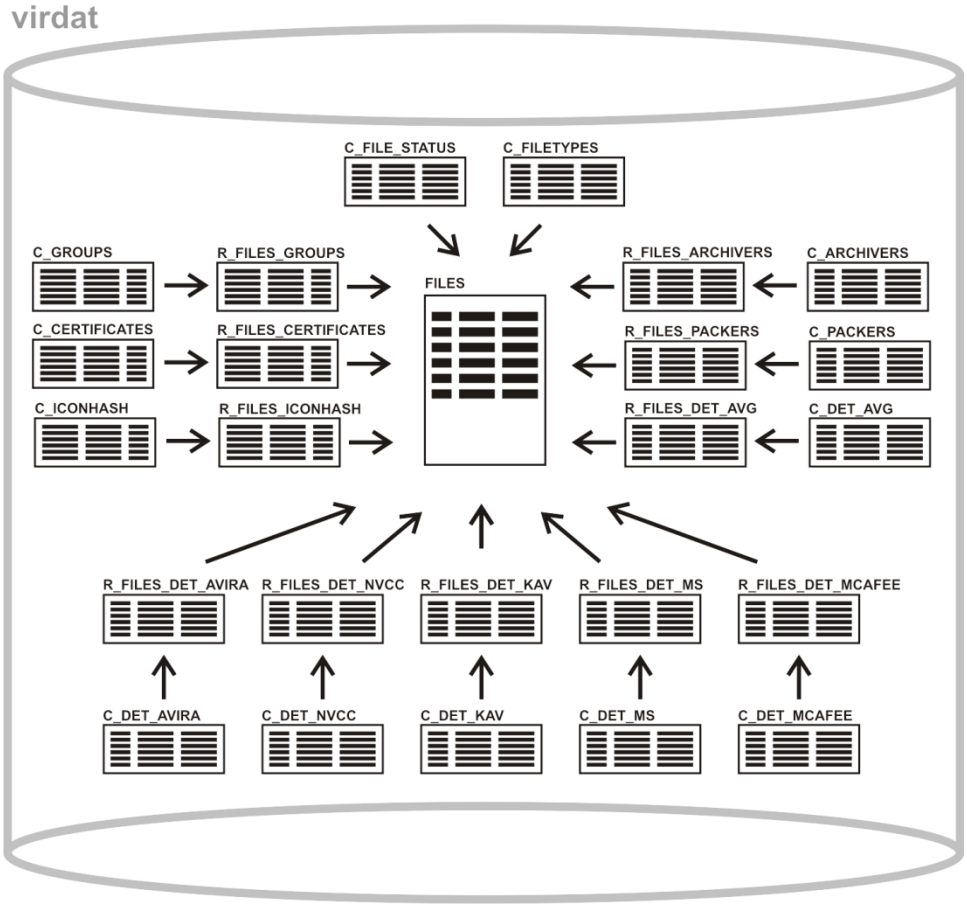
- výrobce databáze: MySQL
    - typ databáze: MySQL 5.1.41 community server
- Databáze virdat:
    - velikost databáze [GB]:
        - data: 35,1 GB
        - indexy 24,9 GB
    - počet tabulek: 86
    - formáty úložišť: InnoDB, MyISAM
- Databázový stroj – hardware
    - procesory: Inte(R) Xeon(R) X5560 o kmitočtu 2,80 GHz a 2,79 GHz (2 procesory)
    - velikost operační paměti RAM: 64 GB

Z výše uvedených hodnot je vidět, že se jedná o poměrně rozsáhlou databázi, ale současně jde i o poměrně výkonný stroj, na kterém je databáze bez problémů spuštěna.

Databáze je zapojena jako komponenta pro různé druhy automatického zpracování. Současně jsou z ní stahována data pro potřeby manuálního zpracování. Data v databázi jsou užívána i za účelem zpřesnění virových definic či za účelem odstranění chybné detekce (angl. „false positive“). A jelikož databáze `virdat` slouží jako sklad informací o všech vzorcích, které měla společnost AVG kdy k dispozici, tak je využívána i jako zdroj pro nejrůznější statistiky.

2.1 Struktura databáze virdat

Na následujících stránkách popíši, jak vypadá struktura databáze `virdat`. Půjde zejména o význam jednotlivých tabulek, které budou použity v následně navrhovaném systému. Jejich detailní popis (typ tabulky, využívané sloupce, velikost tabulek a indexy) bude uveden v kapitole implementace tabulek databáze `virdat`.



Obrázek 1: Struktura databáze virdat – použité tabulky

Tabulka files

Tato databázová tabulka je stěžejním úložištěm pro virové vzorky v rámci virové databáze. Každý záznam reprezentuje jeden vzorek, který společnost AVG Technologies kdy získala. Jak je vidět z obrázku výše, tak zbývající tabulky jsou „pouze“ tabulky „vlastností“ jednotlivých vzorků.

Z tabulky `files` budou využívány následující informace:

first_seen – čas příchodu vzorku – tedy okamžik, kdy vzorek obdržela společnost AVG.

STATUS – status vzorku – jedná se o hodnotu, která popisuje, co o vzorku víme, nebo zda je vzorek teprve zpracováván. Detailní popis jednotlivých hodnot je uveden v podkapitole Tabulka `c_file_status`. Tabulka `c_file_status` je číselníkem (slovníkem) pro tyto hodnoty, jež jsou fakticky identifikátory do tabulky.

filesize – velikost vzorku – `filesize` – v bajtech udávaná velikost obdrženého vzorku.

type & subtype – jedná se o dvojici hodnot, které definují, jakého je vzorek typu. Tzn. jestli je vzorek spustitelný nebo jestli se jedná o knihovnu či jiný typ, který společnost AVG sleduje. Tento sloupce – atribut – je podobně jako status vzorku pouze identifikátorem do tabulky `c_filetypes`, jejíž popis je uveden níže.

secthash – tento atribut je relativně specializovaný a pochází přímo z produkce společnosti AVG. Jedná se o hash ze specifických míst (sekcí) uvnitř souborů Portable Executable. Tento atribut může být významným ukazatelem podobnosti mezi jednotlivými soubory, proto je zahrnut v tomto výčtu a bude používán k vyhledávání v později navrženém systému.

version info CompanyName, FileDescription, FileVersion, LegalCopyright – jedná se o čtveřici hodnot, které mohou obsahovat spustitelné vzorky ve svých zdrojích (vychází z anglického slova `resources`). Obvykle se zde objevuje název výrobce (ať už pravý, či falešný), popis funkčnosti, nebo cokoli, co sem tvůrce programu zapíše.

Tabulky c_archivers a r_file_archiver

Tato dvojice tabulek nese informace o „archivátorech“, tedy o kompresních programech, které rozbalují svůj obsah na disk (`ARJ`, `ZIP`, `RAR`, …). Po strukturální stránce se jedná o dvě tabulky.

První z nich, `c_archivers`, je číselníkem – slovníkem, chcete-li –, kde je uložen seznam těchto programů. Seznam se postupně rozšiřuje, vždy po setkání s novým typem, což umožňuje držet tabulku „přiměřeně“ velikou.

Druhou tabulkou dvojice je `r_file_archiver`. Jak už název naznačuje, jde o tabulku vazební – r jako relation –, přes kterou je možné spojit příslušný archivační software s daným vzorkem tabulky `files`. Tímto způsobem jsme schopni zaručit, že daný vzorek může být archivován různými archivátory. Tento fakt je dále popisován při návrhu systému vyhledávání v sekci požadavků na jednotlivé atributy.

Tabulky c_certificates a r_files_certificates

Tyto tabulky umožňují spojit informace o certifikátech spolu s jednotlivými vzorky, které mají tyto certifikáty uložené v místě zvaném overlay. Overlay je úsek programu, který je nespustitelný, a dal by se popsat jako přílepek daného vzorku. Současně je potřeba zdůraznit, že nejde o položku v sekci `resources`.

Rozeznáváme několik stavů certifikátů. Tyto stavy a schopnost jejich rozpoznání ukazují na důležitost existence těchto informací v databázi, jelikož mohou výrazně pomoci při analýze vzorku. Certifikáty tedy mohou být platné, neplatné, prošlé, ukradené a přitom platné, ukradené a neplatné, vytvořeny k danému souboru a pro něj validní a konečně zcizené z jiného souboru a pro daný vzorek tedy nepatřičné.

Podobně jako u archivátorů, tak i v tomto případě dvojice tabulek znázorňuje slovník a vazební tabulku. Ve slovníku (`c_certificates`) jsou uloženy jednotlivé hashe SHA–256 z certifikátů a další informace, které jsou pro navrhovaný systém irelevantní.

Vazební tabulka certifikátů (`r_files_certificates`) opět umožňuje spojení s jednotlivými vzorky z tabulky `files`.

Tabulky detekcí

Na následujících řádcích je uveden seznam a popis tabulek, které nesou informace o jednotlivých detekcích různých antivirových výrobců. V současné době má společnost AVG k dispozici informace ze sedmi antivirových strojů (angl. engine). Jedná se o antivirové stroje společností AVG, Avira, Kaspersky, McAfee, Microsoft, NOD a Normann Antivirus.

Jelikož každý z antivirových výrobců má svůj speciální přístup, podle kterého detekuje či nedetekuje jednotlivé vzorky (chování, binární signatury, …), může existovat potřeba vyhledávat vzorky podle určitých názvů detekcí nebo určitých antivirových výrobců. Z toho důvodu existuje v databázi `virdat` sada tabulek, které nesou tyto informace.

c_det_avg a r_files_det_avg – Tyto tabulky vytvářejí možnost, jak připojit detekce společnosti AVG s jednotlivými vzorky z tabulky `files`. Jako obvykle se jedná o duo slovník a vazební tabulka.

c_det_avira a r_files_det_avira – V těchto tabulkách jsou uchovány názvy detekcí antivirového stroje společnosti Avira. I zde se jedná o formát slovník (`c_det_avira`) a vazební tabulka (`r_files_def_avira`).

c_det_kav a r_files_det_kav – V těchto tabulkách jsou uchovány názvy detekcí antivirového stroje společnosti Kaspersky. I zde se jedná o formát slovník (`c_det_kav`) a vazební tabulka (`r_files_def_kav`).

c_det_mcafee a r_files_det_mcafee – V těchto tabulkách jsou uchovány názvy detekcí antivirového stroje společnosti McAfee. I zde se jedná o formát slovník (`c_det_mcaffe`) a vazební tabulka (`r_files_def_mcafee`).

c_det_ms a r_files_det_ms - V těchto tabulkách jsou uchovány názvy detekcí antivirového stroje společnosti Microsoft. I zde se jedná o formát slovník (`c_det_ms`) a vazební tabulka (`r_files_def_ms`).

c_det_nod a r_files_det_nod – V těchto tabulkách jsou uchovány názvy detekcí antivirového stroje společnosti NOD. I zde se jedná o formát slovník (`c_det_nod`) a vazební tabulka (`r_files_def_nod`).

c_det_nvcc a r_files_det_nvcc – V těchto tabulkách jsou uchovány názvy detekcí antivirového stroje společnosti Norman Antivirus. I zde se jedná o formát slovník (`c_det_nvcc`) a vazební tabulka (`r_files_def_nvcc`).

Tabulka c_file_status

V kapitole Tabulka `files` je velmi stručný popis sloupce `STATUS`. Tento sloupec prostřednictvím svých hodnot (identifikátorů) ukazuje do „svého“ slovníku, tabulky `c_file_status`. Hodnoty, které tabulka obsahuje, budou nyní stručně popsány:

- **unknown** – vzorek má tuto hodnotu, neprošel-li ještě žádným zpracováním.
- **clean** – jedná se o vzorek, který neobsahuje škodlivý kód – zpracováním tedy již prošel.
- **infected** – vzorek s touto vlastností obsahuje škodlivý kód .
- **unwanted** – vzorek z nějakého důvodu (jedná se o poškozený soubor) nechceme dále zpracovávat.
- **waiting for manual analysis** – vzorky, které je potřeba zpracovat ručně.
- **detection in next update** – vzorky budou detekovány příští aktualizací aplikace AVG.

Každý ze vzorků v tabulce `files` nabývá právě jedné takové hodnoty. Tato hodnota, jak vyplývá z popisů jednotlivých hodnot, se v průběhu zpracování a „života“ uvnitř databáze mění.

Tabulka c_filetypes

V tomto případě se jedná o tabulku typů (slovník) jednotlivých vzorků (`PE`, `DLL`, `script`, aj.). V tuto chvíli databáze `virdat` rozpoznává celkem 22 různých typů souborů. Vztah typu vzorku ke vzorku samotnému je jeden k jednomu. Logicky, daný vzorek může mít pouze jeden typ.

Tabulka c_groups a r_files_groups

Tabulka skupin vzorků. Vzorek může spadat do žádné až několika specifických skupin, které budou popsány níže. Podobně jako v předešlých případech se dá tato vlastnost spojit s tabulkou `files` prostřednictvím vazební tabulky `r_files_groups`.

Zajímavostí pro tento atribut je fakt, že obvykle vzorek nepatří ani do jedné ze skupin a je jen malé procento těch, které do některé patří. Zpravidla vzorky nejsou ve skupině dlouho, jelikož se ve své podstatě jedná pouze o příznaky dalšího zpracování. To je ostatně poznat z popisu jednotlivých skupin níže.

Momentálně máme pět skupin:

- **marx_exclude** – vzorky s touto vlastností jsou v seznamu výjimek pro testování podle Andrease Marxe.
- **foranl** – vzorky, které čekají na ruční zpracování.
- **foranhi** – vzorky, které čekají na manuální zpracování a mají vyšší prioritu.
- **daily_samples** – skupina vzorků, kterou společnost AVG posílá jednou denně svým partnerům.
- **rescan** – vzorky, které budou muset být znovu prověřeny antivirem AVG.

Tabulky c_iconhash a r_files_iconhash

Tabulka `c_iconhash` obsahuje další informace, které můžeme naleznout v sekci ressources u jednotlivých vzorků. Poté, co je ve vzorku nalezena sekce obsahující ikony, je z ní udělána hash – přes celou ikonovou sekci – a tato hash je vložena do tabulky.

Pro spojení s tabulkou `files` je využita vazební tabulka `r_files_iconhash`. A jak již vyplývá z logiky věci, každý vzorek může mít buď žádnou, nebo právě jednu `iconhash`.

Tabulky c_packers a r_files_packer

Posledním zmiňovaným atributem jsou „packery“ (`Bat2Exec`, `Com2Exe`, aj.). Jsou to programy, které dělíme na tří hlavní zástupce.

Prvním z nich jsou programy, které umisťují škodlivý kód do sekce overlay. Takové programy jsou složeny z části pahýlu (angl. stub) programu (`WinRAR Winzipselfextraktory`, `Nullsoft installer`, `autoit`, `BAT2exe`), který není sám o sobě škodlivý, a při spuštění se zavádí právě tento kód. Referenční kód je načten z disku – využívá se toho, že vykonavatelný kód načítaný zavaděčem při spuštění a vlastní soubor na disku mohou mít různé velikosti (kdy soubor na disku je větší).

Druhým zástupcem jsou „runtime packery“, což je v podstatě legální program (`UPX` komprese), který není nijak nebezpečný. Existují „packery“ (`Molebox`, `Themida`, `Bero`), které jsou pro legální software používány jen velmi zřídka. Takto zabalený software je v nejlepším případě na pomezí malwaru. „Runtime packer“ má referenční kód uvnitř vlastního kódu, jím zabalený soubor se zavádí celý a nemá „overlay“. Dekódování probíhá až v paměti po spuštění.

Třetím a posledním zástupcem „packerů“ jsou tzv. „obfuskátory“, tedy programy, které znemožňují automatické a v mnoha případech i ruční zpracování. Technologicky se jedná o „runtime packery“. Kód neoptimalizují, činí ho nepřehledným a brání v jeho pochopení.

Podobně jako u „archivátorů“ je tabulka `c_packers` slovníkem známých „packerů“. Jeden vzorek může být obecně zabalen více různými „packery“, ale obvykle ne současně. To znamená, že vztah mezi jednotlivými „packery“ v takovém případě je vztah OR (buď ten nebo onen) a nikoli AND. Ovšem jsou i specifické případy, kdy vzorky, které jsou typicky legální „stub“ (`SFX`, `bat2exe`,

autoit, NSIS), mají závadnou část v sekci overlay a dále jsou zabaleny opět legálním komerčním UPX „packerem“ (kombinace AND u „packerů“). Zde je cílem zefektivnit prováděný kód a ne znepřehlednit. K realizaci těchto vztahů slouží vazební tabulka r_file_packers. Tento vztah je dále popsán v sekci, která se zabývá požadavky na jednotlivé atributy.

2.2 Implementace databáze virdat

Tato podkapitola si klade za úkol popsat implementaci databáze virdat. Na následujících řádcích tedy jsou mimo jiné uvedeny velikosti jednotlivých, výše diskutovaných, tabulek. Tyto velikosti jsou uvedeny tentokrát v počtech záznamů, nikoli v bajtech. Je to z toho důvodu, aby si čtenář dokázal představit, jaká je přibližná velikost jednotlivých tabulek. Ovšem mnohem důležitější hodnotou je počet ovlivněných vzorků. Tedy počet vzorků, které mohou obsahovat tuto vlastnost.

Vezmeme-li příklad, kdy hledáme vzorky podle atributu STATUS (stav jednotlivých vzorků), tak i přestože stavů je jen pět, tak tuto vlastnost má každý vzorek v databázi. To znamená, že v extrému může být finální množina podstatně větší a hledání (až do vrácení výsledků) bude delší než v případě, kdy hledáme jen na omezeném intervalu vzorků (secthash, která je pouze u vzorku Portable Executable).

Z výše uvedeného odstavce vyplývá, že součástí popisu implementace bude i stručná zmínka o indexování jednotlivých atributů. Tato informace je pro čtenáře důležitým ukazatelem, které sloupce (vyhledávané atributy) jsou indexovány a které ne.

Současně tato část již trochu zasahuje do popisu požadavků na hledané atributy, neboť z tohoto pohledu již lze odhadnout, jestli na vstupu bude pouze jeden nebo více hodnot stejného atributu (na základě velikosti vazební tabulky vůči počtu ovlivněných vzorků). Ovšem tyto požadavky jsou detailně popsány až v kapitole návrhu aplikace, konkrétně v podkapitole Požadavky na hledané atributy.

Pro jednoduchost a přehlednost jsem zvolil uniformní strukturu popisu implementace jednotlivých tabulek. To umožní porovnat jednotlivé tabulky mezi sebou vzhledem k počtu řádků, počtu ovlivněných vzorků, použitému ukládacímu stroji (angl. storage engine) a rozložením indexů.

Hodnoty jsou platné k 26. 1. 2011.

Tabulka files

- počet řádků v tabulce: 29 131 043 řádků
- počet ovlivněných vzorků:
    - sloupec verinfo_companyname: 7 071 805 vzorků
    - sloupec verinfo_filedesc: 6 993 719 vzorků
    - sloupce verinfo_legalCopyright: 7 082 557 vzorků

    - sloupec `verinfo_fileversion`: 8 949 581 vzorků
    - sloupec `secthash`: 21 698 224 vzorků
    - sloupec `first_seen`: 29 131 043 vzorků – celá tabulka
    - sloupec `filesize`: 29 131 043 vzorků – celá tabulka
- engine tabulky: InnoDB
- indexy:
    - sloupec `secthash`
    - sloupec `file_status`

Následující atributy mají indexované jak svoje slovníky (`c_???`), tak i svoje vazební tabulky (`r_file?_???`). Při detailnějším pohledu na indexy u vazebních tabulek zjistíme, že obvykle obsahují dva indexy. První (primární klíč) je buď přes oba sloupce (id bázové tabulky a id slovníku), nebo pouze přes jeden sloupec (id bázové tabulky). Index přes oba sloupce existuje, pokud může daný vzorek nabývat více než jednoho atributu (archivované vzorky – různé archivátory). Index přes jeden sloupec je v případě existence vztahu 1:1 (atribut velikosti souborů).

Současně všechny následující tabulky jsou typu `InnoDB`, protože se využívá při práci s nimi transakcí. Jelikož by se informace v následujícím výčtu neustále opakovaly, budou v popisu ignorovány.

Tabulky c_archivers a r_file_archiver

- počet řádků v tabulce:
    - slovník: 208 řádků
    - vazební tabulka: 5 391 807 řádků
- počet ovlivněných vzorků: 3 617 430 vzorků

Zde je dobře vidět, že ačkoli jeden vzorek může být v daném čase archivován pouze jedním archivátorem, nic nebrání tomu, aby byla kopie toho samého vzorku archivována zcela jiným archivátorem. Proto je mezi pěti a půl milionu archivovaných vzorků pouze tři a půl milionu jedinečných vzorků…

Tabulky c_certificates a r_files_certificates

- počet řádků v tabulce:
    - slovník: 20 089 řádků
    - vazební tabulka: 974 001 řádků
- počet ovlivněných vzorků: 974 001 vzorků

Vhledem k tomu, že soubory jsou v tabulce `files` rozpoznávány pomocí hashe md5 z celého obsahu, je celkem jasné, že jeden vzorek (jedna hash md5) bude mít pouze jeden certifikát (vztah 1:1). Pokud se tento certifikát změní, tak se již změní i hash z příchozího vzorku. Jedná se tedy o jakousi „vnitřní vlastnost“ vzorku, která je spojená přímo s jeho obsahem.

Tabulky detekcí

Jednotlivé tabulky detekcí jsou z implementačního pohledu klasickým vztahem 1:N. Jedna detekce může pokrývat N vzorků. Ovšem podíváme-li se na celou věc ze strany vzorků – tak jak doposud a tak jak tento fakt pojímá i následná aplikace –, pak jeden vzorek má maximálně jednu detekci daného antivirového stroje (nula v případě, že jej daný antivirový stroj nedetekuje).

Odstavec výše platí pochopitelně pro všechny antivirové stroje (tabulky).

Tabulky c_det_avg a r_files_def_avg:

- počet řádků v tabulce:
    - slovník: 4 688 316 řádků
    - vazební tabulka: 21 741 924 řádků
- počet ovlivněných vzorků: 21 741 924 vzorků

Tabulky c_det_avira a r_files_def_avira:

- počet řádků v tabulce:
    - slovník: 2 376 449 řádků
    - vazební tabulka: 14 992 945 řádků
- počet ovlivněných vzorků: 14 992 945 vzorků

Tabulky c_det_kav a r_files_def_kav:

- počet řádků v tabulce:
    - slovník: 4 688 316 řádků
    - vazební tabulka: 14 754 660 řádků
- počet ovlivněných vzorků: 14 754 660 vzorků

Tabulky c_det_mcafee a r_files_def_mcafee:

- počet řádků v tabulce:
    - slovník: 680 027 řádků
    - vazební tabulka: 15 753 097 řádků
- počet ovlivněných vzorků: 15 753 097 vzorků

Tabulky c_det_ms a r_files_def_ms:

- počet řádků v tabulce:
    - slovník: 44 572 řádků
    - vazební tabulka: 3 978 745 řádků
- počet ovlivněných vzorků: 3 978 745 vzorků

Tabulky c_det_nod a r_files_def_nod:

- počet řádků v tabulce:
    - slovník: 336 707 řádků
    - vazební tabulka 13 329 915 řádků
- počet ovlivněných vzorků: 13 329 915 vzorků

Tabulky c_det_nvcc a r_files_def_nvcc:

- počet řádků v tabulce:
    - slovník: 4 312 106 řádků
    - vazební tabulka: 13 011 697 řádků
- počet ovlivněných vzorků: 13 011 697 vzorků

Tabulka c_file_status

- počet řádků v tabulce:
    - slovník: 5 řádků
    - vazební tabulka: neexistuje, je přímo napojená na tabulku files
- počet ovlivněných vzorků: 29 131 043 vzorků

Jak již bylo uvedeno v popisu struktury databáze, každý vzorek má svůj status, takže se jedná o vztah 1:1. Identifikátor vzorku je uložen přímo v tabulce `files`, takže tento atribut „postrádá“ vazební tabulku – z implementačního pohledu ji není třeba.

Tabulka c_filetypes

- počet řádků v tabulce:
    - slovník: 22 řádků
    - vazební tabulka: neexistuje, je přímo napojena na tabulku files
- počet ovlivněných vzorků: 29 131 043 vzorků

Typ souboru (file type) je přesně ten samý případ jako stav souboru (file status).

Tabulka c_groups a r_files_groups

- počet řádků v tabulce:
    - slovník: 5 řádků
    - vazební tabulka: 1 919 625 řádků
- počet ovlivněných vzorků: 1 919 503 vzorků

U skupin jde o vztah 1:N. Jeden vzorek se může nacházet ve více skupinách, byť tento stav nemusí trvat dlouho. Proto je zde tak malý rozdíl mezi počtem ovlivněných vzorků a velikostí vazební tabulky. Jinými slovy: počet vzorků, které jsou momentálně ve více třídách, je rozdíl mezi jednotlivými hodnotami.

Tabulky c_iconhash a r_files_iconhash

- počet řádků v tabulce:
    - slovník: 1 515 766 řádků
    - vazební tabulka: 11 312 275 řádků
- počet ovlivněných vzorků: 11 312 275 vzorků

S hashí z oblasti ikon je situace naprosto stejná jako s certifikáty. Opět se jedná o vnitřní vlastnost daného vzorku (hash je pořízena z celé sekce ikon v sekci ressource ve vzorku), takže vztah 1:1 je naprosto evidentní.

Tabulky c_packers a r_files_packer

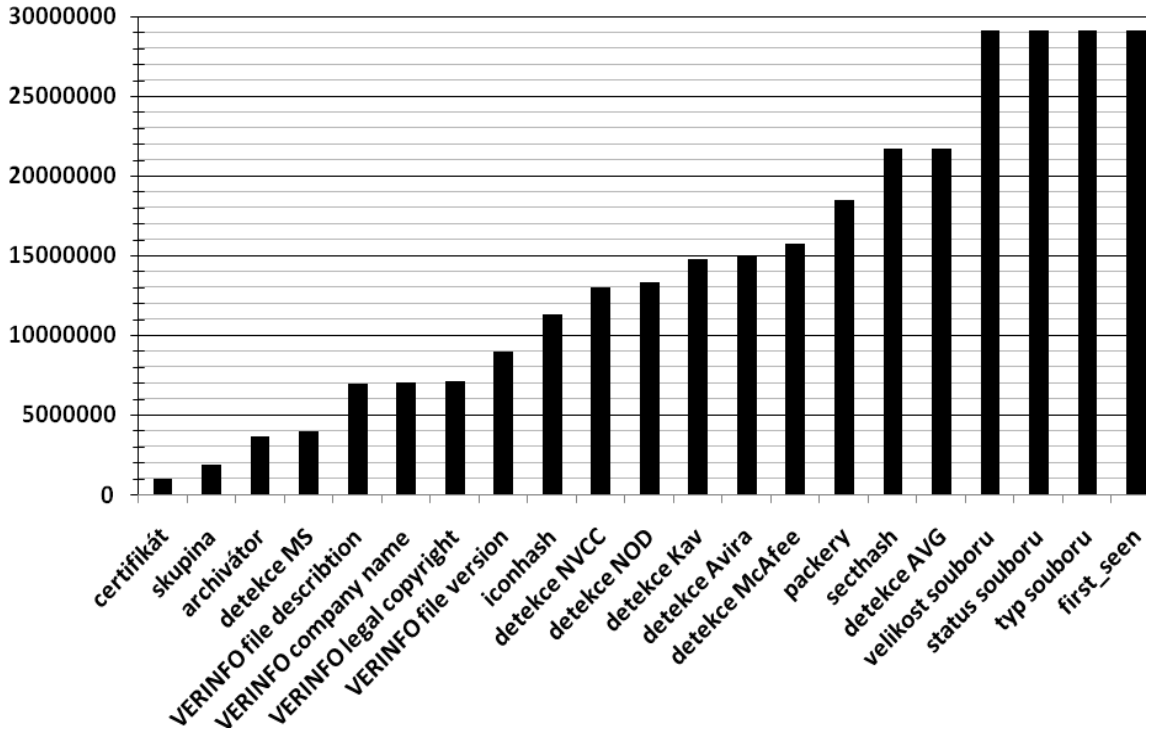
- počet řádků tabulce:
    - slovník: 494 řádků
    - vazební tabulka: 26 365 739 řádků
- počet ovlivněných vzorků: 18 516 999 vzorků

„Runtime packery“ jsou prakticky, co se implementace databáze týče, naprosto totožné s archivátory.

Níže uvedené grafické znázornění pak ukazuje, jaké atributy budou mít největší vliv na jednotlivé vzorky. Z pohledu návrhu jednotlivých dotazů pro atributy: Pokud budeme vyhledávat podle atributů, které mají vyšší počet ovlivněných vzorků, pak nejspíše hledání dosáhne delších časů než u atributů s menší „ovlivnitelností“. Je tomu tak proto, že se bude muset projít více vzorků, byť po indexu. Tato myšlenka je více rozvedena v návrhu systémů vyhledávání dále v diplomové práci.

Poslední odstavec této kapitoly bych chtěl věnovat typu databáze `virdat`. Někteří zlí jazykové totiž říkají, že databázový stroj MySQL je vhodný pouze pro malé a střední databáze. `Virdat` dle mého názoru není ani jedno. Celkový počet řádků v tabulce `files` je k dnešnímu datu

(20. 10. 2011) 41 890 416 a to je pouze jedna tabulka, byť ta největší. Současně je na serveru spuštěno těchto databází několik. Dále bych ještě uvedl, že se jedná o volně dostupnou verzi databáze, tedy takovou, kterou si každý z nás může stáhnout a používat pro nekomerční účely. Myslím si, že výše uvedená fakta dostatečně motivují čtenáře k používání této databáze pro větší nekomerční projekty.



Obrázek 2: Grafické znázornění atributů a počtu jimi ovlivněných vzorků

3 Možnosti optimalizace dotazů nad rozsáhlými databázemi

Následující kapitola se zabývá možnostmi, jak tvořit efektivní dotazy pro databáze MySQL. Bude zde uvedeno několik dobrých rad, převážně pocházejících z [1] a samozřejmě také z webových stránek výrobce, konkrétně z [2]. Bude zde popsáno, jakým konstrukcím se vyhnout a jaké konstrukce místo těch problematických zakomponovat, aby byl dotaz co možná nejefektivnější. Současně musím zdůraznit, že jelikož je celá tato problematika velmi rozsáhlá, tak zde budu diskutovat pouze některé optimalizace dotazů. Některé z těchto rad jsem později využil při tvorbě vyhledávacího systému. Všechny tyto informace budou začleněny do první podkapitoly a ještě zde bude dále uvedeno i několik informací o současných schopnostech optimalizátoru MySQL.

Druhá podkapitola se bude zabývat problematikou, jak měřit rychlost kladených dotazů, a na konci této podkapitoly bude důkladně popsán nástroj `QueryCounter`, který jsem vytvořil, abych mohl testovat náročnost jednotlivých konstrukcí SQL.

3.1 Optimalizace SQL dotazů pro databáze MySQL

Pokud není uvedeno jinak, jsou informace obsažené v rámci této kapitoly převzaty z [1].

3.1.1 Optimalizace přístupu k datům

Nadbytečné množství nepotřebných dat je jeden z hlavních důvodů špatného výkonu dotazu. Povětšinou nepotřebujeme, a vzhledem k časové náročnosti je dokonce velmi nežádoucí, aby databázový stroj procházel celou tabulku. Z tohoto důvodu je dobré řídit se následujícími třemi radami, které se zaměřují na optimalizaci přístupu k datům.

Omezení počtu řádků

Databáze MySQL poskytuje výsledky na vyžádání. To znamená, že klauzule `SELECT` bez jakékoli limitace stáhne opravdu kompletní výsledek dotazu. Proto potřebujeme-li do své aplikace určitý počet výsledků z daného dotazu, měli bychom tento dotaz omezit. K tomu slouží v jazyce SQL klauzule `LIMIT`.

Po technické stránce klient MySQL získá kompletní data, která mu server může na základě dotazu nabídnout, a tato data jsou pak poskytnuta aplikaci, která již s nimi může pracovat. Server MySQL tedy posílá data najednou, nikoli až je aplikace vyžaduje.

Omezení počtu sloupců

Podobně jako omezení počtu řádků je i omezení počtu sloupců. Opět je třeba si položit otázku, jestli se nedá dotaz nějak rozumně optimalizovat a jestli konstrukce `SELECT *` je opravdu potřeba. Při dotazech se spojením touto konstrukcí získáme všechny sloupce všech tabulek, které ve spojení figurovaly, což při komplikovaných dotazech může vyústit k velkému množství dat a dlouhé čekací době. A to i za předpokladu, že jste užili první pravidlo a rozumně jste použili klauzuli `LIMIT`.

Abych ještě zvýšil motivaci, tak bych chtěl dodat, že konstrukce `SELECT *` může potlačit různé jiné optimalizace na databázovém stroji MySQL jako jsou „pokrývací indexy“ [1]. To v důsledku znamená, že doba strávená nad databází se ještě prodlouží…

3.1.2 Restrukturalizace dotazu

Cílem restrukturalizace dotazu je naleznout stejný výsledek (ne nutně stejnou sadu dat) pomocí jiné konstrukce SQL. Ekvivalentní dotaz může být efektivnější, pokud budeme v restrukturalizování úspěšní.

Málo složitých dotazů versus mnoho jednodušších

V tradičním přístupu k databázovému schématu se v minulosti uvádělo, že je velmi žádoucí docílit co nejmenšího počtu dotazů. Dle [1] byl tento přístup opravdu lepší kvůli nákladům na síťovou komunikaci, analýzu a optimalizaci dotazu.

Dnes již tento názor tak striktně neplatí jako v dobách minulých, protože databázový stroj MySQL prošel za tu dobu značným vývojem. Nicméně si musíme uvědomit, že odezva připojení je stále velmi pomalá v porovnání s počtem řádků, které je databázový stroj MySQL schopen interně projít – zvláště pokud jsou data již uložena v paměti a ne na disku. Na druhou stranu je dle [1] velký počet dotazů považován za chybu návrhu aplikace.

Z výše uvedených odstavců tedy vyplývá, že je na tvůrci aplikace, aby zvážil možnosti prostředí a možnosti databázového stroje. Ideálním způsobem, jak získat efektivní variantu, je vytvořit obě možnosti a poté jednoduše srovnat rychlosti zpracování.

Salámová metoda

Toto interní pojmenování dle [1] označuje metodu, kdy neprovádíme dotaz najednou, ale dělíme ho na „menší porce“. Autor zde uvádí příklad na situaci odmazávání starých záznamů z databáze. Pokud se na tuto problematiku podíváme v detailu, tak uvidíme, že než provést dotaz (popsaný níže) jako celek,

```
DELETE FROM table
      WHERE created < DATE_SUB(NOW(), INTERVAL 3 MONTH);
```

SQL dotaz odstraňující staré zprávy na jednou

je lepší ho rozsekat na právě výše zmíněné „menší porce“ a zapouzdřit jeho vykonávání do cyklu, kdy kontrolujeme, jestli daný dotaz ještě „ovlivňuje“ nějaké řádky v tabulce. Pokud tomu už tak není, pak je možné cyklus ukončit, protože se již nic více mazat nebude.

```
DELETE FROM table
      WHERE created < DATE_SUB(NOW(), INTERVAL 3 MONTH)
      LIMIT 10000;
```

Dotaz SQL odstraňující zprávy po částech

Jak dále popisuje autor [1], tak tato metoda je výhodnější, protože limitace na 10 000 řádků už je dosti rozsáhlá, aby se každý dotaz vykonal efektivně, a současně se snižuje dopad na server (transakční úložné enginy mohou těžit z menších transakcí). Poslední věcí, kterou autor [1] uvádí k této problematice, je vložit mezi jednotlivé dotazy nějakou menší prodlevu, aby se zátěž lépe rozložila v čase a zredukovala se doba uzamykání jednotlivých tabulek.

Dekompozice spojení

Podstata této metody spočívá v rozložení velkého dotazu, který sestává z několika vnitřních spojení, do série jednotlivých dotazů bez spojení. Za spojení jednotlivých výsledků tedy již nebude odpovědná databáze, ale aplikace, která dotazy pokládá.

```
SELECT * FROM tab1
      JOIN tab2 ON tab2.tab1_id = tab1.id
      JOIN tab3 ON tab3.tab2_id = tab2.id
WHERE  tab1.content = 'whatever';
```

dotaz SQL před dekompozicí

```
SELECT * FROM tab1 WHERE tab1.content = 'whatever';
SELECT * FROM tab2 WHERE tab2.tab1_id = 1234;
SELECT * FROM tab3 WHERE tab3.tab2_id IN (1,2,3,4);
```

dotaz SQL po dekompozici

Ačkoli je tento postup v rozporu s předchozí optimalizací, tak autor [1] uvádí pět důvodů, proč může tento dotaz dosahovat lepších výsledků z pohledu výkonu. Tyto body budou stručně popsány. Pro detailnější popis jednotlivých aspektů doporučuji přečíst stranu 184 v [1].

- Ukládání do mezipaměti může být efektivnější – při opakovaném hledání konkrétní hodnoty se některé dotazy mohou přímo přeskočit.

- Efektivnější vykonávání zámků nad tabulkami typu MyISAM.

- Lepší škálovatelnost – tabulky se mohou umístit na různé servery.
- Použití klauzule IN je v databázovém stroji MySQL efektivnější – bude diskutováno dále v této kapitole.
- Redukují se redundantní přístupy k řádkům.

3.1.3 Současné schopnosti optimalizátoru MySQL

V následujících odstavcích jsem vybral několik optimalizací z [1], které je databázový stroj MySQL schopen provádět sám v rámci svého běhu. Pro detailnější popis a některé další optimalizace nahlédněte na stranu 190 v [1]. Je však třeba uvést, že ani [1] neuvádí všechny dostupné optimalizace, které databázový stroj MySQL dokáže provádět.

Optimalizace COUNT(), MIN(), MAX()

Existuje-li nad sloupcem index a je-li znám fakt, zda daný sloupec může nabývat hodnot `NULL`, pak je schopen optimalizátor databázového stroje MySQL velmi jednoduše dohledat nejmenší hodnotu (`MIN`) a největší hodnotu (`MAX`). Optimalizátor to provede tak, že se podívá na první či poslední záznam daného indexu. Pokud je tento index typu `B-TREE`, pak se podívá na hodnoty nejvíce vlevo, potažmo nejvíce vpravo.

Tato optimalizace může být vykonána ještě před prováděním vlastního dotazu, takže se s výsledky funkcí `MIN` a `MAX` bude pracovat jako s konstantami.

Optimalizace funkce `COUNT` je trochu specifičtější, ale v důsledku stejná. Aby byla optimalizace před provedením dotazu úspěšná, tak v dotazu musí chybět klauzule `WHERE`. Současně je tato optimalizace dostupná pouze u některých úložných strojů (třeba stroj MyISAM má přesný přehled o počtu všech svých řádků). V takové případě se opět výsledek funkce `COUNT` promítne do dotazu jako konstanta.

Aplikace ekvivalentní algebraických pravidel

Současný optimalizátor databázového stroje MySQL dovede zjednodušovat algebraický výrazy. Mezi aplikované optimalizace patří „seškrtání konstant“, eliminace nemožných omezení a konstantní podmínky.

Vyhodnocování a redukce konstantních výrazů

Jak již bylo výše uvedeno, optimalizátor databázového stroje MySQL je schopen provádět redukce na konstanty. Tyto redukce je možno provádět jak z určitých funkcí (`MIN`, `MAX`,`COUNT`), tak i z uživatelských proměnných, pokud se nemění v čase. Redukují se i určité aritmetické výrazy.

Ovšem optimalizátor je schopen redukovat i některé celé dotazy. Provádíme-li hledání nad jedinečným indexem nebo hledáme-li na primárním klíči a současně máme-li v klauzuli `WHERE`

konstantu (konstantní podmínku), pak je možná redukce tohoto celého dotazu na konstantu. Pro lepší představu jsem převzal jeden konkrétní příklad z [1].

```
SELECT tab1.tab1_id, tab2.tab2_id
      FROM tab1
      INNER JOIN tab2 USING(tab1_id)
      WHERE tab1.tab1_id = 1;
```

Brzké ukončení výrazů

Databázový stroj MySQL dovede ukončit dotaz v okamžiku, kdy jsou splněny parametry, které krok nebo dotaz požadoval. Nejjednodušším příkladem je klauzule `LIMIT`. Současně ale existují i další možnosti, kdy server ukončí dotaz předčasně.

Jednou takovou situací je identifikace nemožné podmínky (třeba záporné číslo v bezznaménkovém sloupci). V takovém případě je dotaz zastaven ještě ve fázi optimalizace. Autor [1] uvádí, že takové optimalizace jsou implementovány i v některých druzích dotazů s `DISTINCT`, `NOT EXISTS` a `LEFT JOIN`.

Porovnání v seznamech IN()

Hodnoty z výčtu `IN()` jsou v mnohých databázových systémech ekvivalentem několika za sebou sdružených klauzulí `OR`. Databázový stroj MySQL je ovšem v tomto ohledu jiný. Hodnoty se ve výčtu seřadí a použije se rychlé binární hledání, které zjistí, jestli je hodnota v seznamu či nikoli.

Seznamy (výčet hodnot uvnitř `IN`) by neměly být příliš dlouhé, protože pak je zpracovávání podstatně delší, než u dotazů s více klauzulemi `OR`. Dle [1] je náročnost operace s klauzulí `IN` logaritmická `O(log(n))`, kdežto při zpracování s klauzulemi `OR` je náročnost lineární `O(n)`.

3.1.4 Optimalizace specifických typů dotazů

V následujících sekcích bych rád popsal některé optimalizace pro specifické typy dotazů. Obsahy a typy optimalizací pocházejí ze strany 215 v [1].

Optimalizace dotazu s COUNT()

Dotazy s klauzulí `COUNT` jsou velmi obtížně optimalizovatelné, protože musejí projít velké množství řádků ve velkých tabulkách. Existují varianty, kdy je tento dotaz převeditelný na konstantu (viz výše v této kapitole), ale jestli tato optimalizace není proveditelná, pak zbývá pouze použití pokrývacího indexu, který důkladně popsán v kapitole 3 v [1].

Současně pokud ani tato metoda není dostatečně účinná, pak je třeba použít jiných technik, které nejsou součástí vykonávání databázového dotazu. Mezi tyto techniky autor [1] zařadil třeba sumarizační tabulky, jejichž účel je schraňovat data (byť třeba ne zcela aktuální) o velikostech

tabulek či výsledky některých často používaných agregačních funkcích. Dalším „uměle“ vytvořeným mechanismem může být externí systém fungující jako mezipaměť.

Optimalizace dotazů s JOIN

U optimalizací dotazů s klauzulí `JOIN` je potřeba se pokusit dodržet následující zásady pro tvorbu dotazů. Sloupce, které jsou při spojení uvedeny za klauzulí `ON` či v klauzuli `USING`, musí být indexované. Současně zvažujte pořadí spojovaných tabulek. Pokud podle příkladu na straně 217 v [1] spojujeme tabulky v pořadí A a B a rozhodne-li optimalizátor, že spojení provede v pořadí B a pak až A, tak není třeba indexovat sloupec v tabulce B. Pokud už index nad daným sloupcem v tabulce B existuje z jiných důvodů, pak jej zachováme, ale jestli index potřebujeme jen kvůli tomuto spojení, pak by měl být odstraněn. Je to z toho důvodu, že nepoužívané indexy zvyšují režii.

Všechny výrazy v klauzulích `GROUP BY` nebo `ORDER BY` by měly pocházet ze sloupců v jedné tabulce. Jedině, tak existuje naděje, že databázový stroj MySQL se pokusí použít index na tyto operace.

Upgrade verze databáze je další věcí, která může výrazně omezit optimalizace stávajícího optimalizátoru. Omezit ve smyslu změnit jeho chování. Běžná spojení mohou dostat povahu kartézských součinů, mohou se změnit syntaxe spojovacích operací nebo může dojít ke změně předností operátorů. Byť jsou předešlé příklady pouze hypotetické, tak je z uvedeného snad jasné, že to, co bylo optimální pro dřívější verzi, již nemusí být optimální v aktuální verzi.

Optimalizace poddotazů

Co se poddotazů týče, radí autor, přepsat je v každém možném případě na pouhá spojení. Je tomu z důvodu nedostatečné optimalizace ze strany databázového stroje MySQL. Byť, jak v [1] popisuje, existuje velká snaha o vylepšení optimalizace poddotazů, nejbližší hmatatelný důkaz bude až v další verzi databáze (aktuální verze pro [1] je verze 5.1).

Optimalizace GROUP BY a DISTINCT

O tom, jak je optimalizátor schopen použít index v případě klauzule `GROUP BY`, jsem se zmiňoval o několik odstavců výše (optimalizace dotazů s `JOIN`). V této sekci ještě ale přidám několik možností, které se dají použít v případě, kdy není možné jít po indexu. Klauzule `GROUP BY` použije pro setřídění buď dočasnou tabulku, nebo metodu `filesort`. Není přesně definované, který způsob je efektivnější, to obvykle záleží na dané situaci. Takže pro efektivní zpracování dotazu je potřeba vyzkoušet obě varianty třídění.

První varianta (dočasná tabulka) se použije v okamžiku, kdy použijeme v dotazu optimalizační pokyn `SQL_BIG_RESULT`. Druhá metoda (`filesort`) se vynutí optimalizačním pokynem `SQL_SMALL_RESULT`.

Optimalizace LIMIT a OFFSET

Prvním krokem pro optimální dotaz s klauzulí `LIMIT` je existence indexu, který podporuje řazení nad vyhledávaným sloupcem (zvláště tedy v případě, že se bude jednat třeba o problematiku stránkování s potřebou řazení).

Dalším způsobem, jak zvýšit výkonnost dotazu, je snížit offset (tedy zmenšit počet stránek v aplikaci). Problematika velkého množství stránek spočívá v tom, že prvotní offset dotazu uvedeného níže vybere z databáze celkem 1010 záznamů a prvních 1000 záznamů vyhodí a zobrazí vámi požadovaných 10 položek (počet položek na stránce sto první stránce).

```
SELECT tab1.id tab1.textCollumn
      FROM tab1
      ORDER BY tab1.collumn2
      LIMIT 10000,10;
```

Pokud z jakéhokoli důvodu není možné počet stránek zmenšit, tak autor uvádí ještě možnost využití offsetu na „pokrývacím indexu“ (pokud existuje). To v podstatě znamená, že nehledáme kompletní sadu dat v daném dotazu (tak jak je tomu v dotazu uvedeném výše), ale vytvoříme poddotaz, který se ptá pouze na identifikátor (sloupec id). To umožní použít „pokrývací index“. Výsledek poddotazu spojíme s požadovaným výčtem sloupců (tak jak uvádí příklad níže).

```
SELECT tab1.id, tab1.textCollumn
FROM tab1
INNER JOIN (
      SELECT id
      FROM tab1
      ORDER BY tab1.collumn2 LIMIT 1000,10
) AS subQuery USING (id);
```

Důvod, proč tato SQL konstrukce bude efektivnější než konstrukce předchozí, je ten, že omezíme přístup k datům v náročné části dotazu. Tedy nebudeme z databáze stahovat tisíc a deset plnohodnotných řádků, ale jen tisíc a deset hodnot typu identifikátoru, a to ještě za pomocí pokrývacího indexu (čili velmi rychle). Plnohodnotné řádky (obsahující požadovaná data) budeme stahovat pouze v těch deseti případech, které zobrazujeme na stránce.

3.1.5 Pokyny pro optimalizátor MySQL

V některých situacích se může přihodit, že programátor není spokojen s vykonávacím plánem, který mu optimalizátor databáze nabídne. Proto existuje sada klauzulí, které dokážou optimalizátor navést tím správným, nebo alespoň podle vás tím správným směrem.

Opět zde uvedu jen několik z těchto klauzulí. Více těchto „pokynů" je k nalezení na straně 222 v [1]. Kompletní seznam je k nalezení na [2]. Ještě před tím, než na následujících odstavcích popíši několik pokynů pro optimalizátor, je třeba zdůraznit, že některé z pokynů jsou závislé na verzi databázového stroje. Proto je nejjistější variantou ověřit validitu daného pokynu na webu výrobce v sekci optimalizací [2] pro danou verzi databáze.

HIGH_PRIORITY, LOW_PRIORITY

Pokud přistupujeme několika SQL dotazy k jedné tabulce, pak výše uvedené klauzule říkají, který dotaz dostane přednost a který naopak zůstane poslední. Ve své podstatě se jedná o možnost předběhnutí ve frontě (`HIGH_PRIORITY`), ve které jednotlivé příkazy čekají (třeba na uvolnění zámku nad danou tabulkou).

V opačném případě (`LOW_PRIORITY`) se daný příkaz zařadí na konec fronty. I když je na konci fronty po příchodu (poslední příbytek do fronty), tak poslední zůstává, dokud bude existovat dotaz (příkaz) se zájmem o danou tabulku.

Tyto příkazy jsou účinné pouze pro úložné stroje (angl. storage engines), které mají uzamykání na úrovni celých tabulek. Je zcela nevhodné je používat všude tam, kde je jemnější schéma zámků (InnoDB – zámky na úrovni řádků). Současně mohou omezit souběžnost vkládání řádků a tím velmi omezit celkový výkon databáze.

STRAIGHT_JOIN

Tento příkaz má za úkol potlačit spojování tabulek tak, jak jej navrhl optimalizátor MySQL. Může se vyskytnout na dvou pozicích v dotazu `SELECT`. První pozicí je místo hned za klauzulí `SELECT`, druhou pozicí je oblast mezi dvěma spojovanými tabulkami.

První způsob použití si vynucuje spojení tabulek tak, jak ho vytvořil programátor (přesně v tomto pořadí). Druhý způsob nutí databázový stroj MySQL spojit ty dvě tabulky, u kterých je klauzule uvedena.

Obecně platí pro tuto klauzuli dvě zásady. První říká, že bychom ji měli užít jen v případě, že není spojení výkonově efektivnější. To znamená, že víme, že naše verze je efektivnější. Druhá zásada spočívá v kontrole dotazů s klauzulí `STRAIGHT_JOIN`, kdykoli je proveden upgrade databáze. Nelze totiž zaručit, že optimalizátor ve vyšší verzi databáze již vaši verzi dotazu bude provádět efektivněji a vy ji klauzulí `STRAIGHT_JOIN` opět potlačíte. Proto by měly být takové dotazy opět testovány.

SQL_SMALL_RESULT, SQL_BIG_RESULT.

Jak již bylo popsáno výše v tomto textu, příkazy `SQL_SMALL_RESULT` a `SQL_BIG_RESULT` se uvádějí jen při příkazu `SELECT` a říkají, kdy se má použít dočasné tabulky a kdy třídění `filesort`. Pro úplnost a plynulost textu zde opět uvedu výše zmíněné optimalizace.

SQL_SMALL_RESULT říká, že očekávaná výsledná sada bude malá, a proto je možné ji uložit do indexované dočasné tabulky. Tuto tabulku nebudeme umisťovat na disk, ale přímo do paměti, čímž získáme podstatně rychlejší přístup k výsledku.

Oproti tomu pokyn SQL_BIG_RESULT říká, že očekáváme velký výsledek, takže bude lepší dočasnou tabulku uložit na disk a použít třídění.

SQL_CACHE, SQL_NO_CACHE

Tento příkaz vysílá databázovému serveru informaci o tom, jestli dotaz, který jej nese, je anebo není určen (doslova jestli je anebo není kandidátem) na ukládání do mezipaměti. Speciálně klauzule SQL_NO_CACHE je hojně využívána pří testování dotazů. Více informací o testování dotazů a o důvodu použití této klauzule jsem uvedl v následující podkapitole.

USE_INDEX, IGNORE_INDEX, FORCE_INDEX

Jak již vyplývá z názvu klauzulí, budou se tyto příkazy týkat indexů. První dvě klauzule nutí optimalizátor použít anebo ignorovat daný index. Ve verzí databázového stroje MySQL 5.0 a dřívější se jedná jen o spojení tabulek. Podle [1] se použití těchto klauzulí nedotkne procedur setřiďování a seskupování. V databázovém stroji MySQL 5.1 se dají pro tyto účely uvést klauzule FOR ORDER BY nebo FOR GROUP BY.

Klauzule FORCE_INDEX je podobná jako USE_INDEX, ale zvyšuje váhu použití zadaného indexu. Jinými slovy, klauzule FORCE_INDEX říká databázovému stroji MySQL, že hledání napříč tabulkou, bez využití zadaného indexu (bez indexu či za použití jiného indexu), bude tak náročné, že by mělo být použito pouze za předpokladu, že není jiné varianty, jak hledaný řádek získat.

3.2 Testování výkonu dotazu

Předchozí podkapitola se zabývala jednotlivými optimalizacemi. Byly tam popsány různé možnosti, jak dosáhnout efektivního zpracování vhodnou konstrukcí dotazů SQL. Tato kapitola si ovšem klade za úkol ukázat techniky měření výkonu dotazu, které umožňují potvrdit či vyvrátit efektivnost vašich konstrukcí v jazyce SQL.

Současně zde bude diskutován i software mé vlastní výroby, který je na některých těchto technikách vystavěn. Popisovány budou především možnosti softwaru v oblasti různých typů měření dotazů. Dále pak zde bude popsán i formát vstupu, který software používá, a v poslední části této podkapitoly bude ještě popsána implementační stránka.

V jednotlivých odstavcích níže jsou uvedeny metody, kterými je možné měřit výkon dotazu. Samozřejmě že se nejedná o kompletní výčet metod pro testování výkonu, ale jen o zlomek. Navíc jsou poměrně jednoduché, a tudíž není problém tyto metody implementovat anebo je jen využít (viz záznam pomalých dotazů).

3.2.1 EXPLAIN

V tomto případě se fakticky nejedná o metodu testování jako spíš o analýzu, kterou nám databázový stroj MySQL nabídne. Stroj MySQL umožní programátorovi dotazu nahlédnout do vykonávacího plánu, kterým se bude ubírat při plnění požadavku na výsledek.

Už zde můžeme získat klíčové informace o tom, jestli bude dotaz alespoň nějak efektivní, jestli případné spojování tabulek půjde po indexu, jestli se budou provádět optimalizace dotazu či jestli je možnost redukcí některých případných algebraických výrazů. Současně jsme schopni zjistit, jestli některý z potenciálních poddotazů je server schopen redukovat na konstantu. Všechny tyto faktory jsou úzce spojeny s časem vykonávání dotazu, ostatně jak již je popsáno v předešlé podkapitole.

Pokud jsou všechny odpovědi na předchozí „jestli“ záporné, tak máme přinejmenším možnost k zamyšlení, proč tomu tak je. Vyvstanou nové otázky, třeba jestli jsou tabulky databáze dostatečné indexovány nebo zda netvoříme vyloženě neefektivní dotaz. V takovém případě nám testování výkonu odpoví pouze na otázku, jak dlouho budeme čekat na tento „nejhorší možný scénář“. Samozřejmě i tento dotaz se počítá a při měření efektivnosti a rychlosti vykonávání nám dává pomyslné dno. Výsledky měření dalších dotazů – tentokráte již efektivnějších – nám mohou poukázat na míru zlepšení.

3.2.2 Čas

První a asi nejjednodušší variantou, jak testovat databázové dotazy, je měření doby jejich vykonávání. Tato metoda vyžaduje dostatečně „jemné“ stopky (sekundy a někdy i milisekundy jsou příliš hrubé pro rychlé dotazy).

Jelikož se dotaz měří od počátku vykonávání (tedy od okamžiku, kdy jej do databáze odešleme) až po jeho dokončení (výsledná sada je zaslána z databáze a připravena na zpracování v aplikaci), tak nám stačí jedny stopky. Na počátku spustíme a na konci zastavíme. Výsledkem je rozdíl těchto dvou časů.

Tato metoda ovšem není moc vhodná pro velmi rychlé a rychlé dotazy. Je to ze dvou důvodů. První z nich je nutnost, aby programové stopky měly dostatečně malý rozdíl mezi jednotlivými tiky (třída pro čas v jazyce C# měří jednotlivé tiky v řádu stovky nanosekund).

Druhým důvodem je umístění dat. Představme si, že jsou data uložena na pevném disku. Bude trvat jistou, relativně dlouhou dobu (v porovnání s vykonáváním vloženého dotazu SQL), než data nahrajeme do paměti (ne všechny databáze světa mají celý svůj obsah již rozbalený v paměti). Položíme-li stejný dotaz na data, která jsou již v paměti, získáme poměrně velký časový rozdíl. Tento rozdíl již může degradovat čas měření, i když daný dotaz je ve skutečnosti velmi rychlý. Pokud i tak chcete využívat tuto metodu na rychlé dotazy, pak zkuste volat tento dotaz vícekrát za sebou a jejich čas zprůměrovat.

Naopak pro velmi pomalé a pomalé dotazy je měření časem vhodné. Zde nebude záležet na tom, jsou-li data v paměti, či nikoli. Může to být z důvodů, že hledáme na tak rozsáhlé množině dat, že se do paměti naráz nikdy nevejdou všechny. Nebo je dotaz příliš komplexní, takže jednoduše nebude možné jej provést rychle ani za předpokladu, že všechna data v paměti jsou.

3.2.3 QPM

Výraz QPM je zkratkou anglického „Queries Per Minute“. Jak již z názvu vyplývá, jedná se o testování počtu dotazů za nějaký čas (za minutu). Tato metoda spočívá v opakovaném zasílání stejného dotazu databázovému serveru.

Podobně jako v předchozím případě se zašle dotaz do databáze, ale po obdržení výsledku se tento dotaz zasílá znovu. S prvním dotazem se spustí i stopky, které měří čas. V okamžiku kdy na stopkách máme minutu (či jiný testovaný časový interval), počkáme na dokončení aktuálně zaslaného databázového dotazu a cyklus ukončíme. Samozřejmě s každým dokončeným dotazem musíme navýšit hodnotu na počítadle dotazů.

Výsledkem tohoto typu testů je tedy počet zaslaných dotazů, na které databáze odpověděla v testovaném časovém intervalu. Z tohoto důvodu se nevyplatí testovat dotazy, o kterých víme, že budou časově náročné. Výsledkem, pokud vykonávací doba dotazu překročí jednu minutu, by byla jednička. V takovém případě budeme tedy pouze vědět, že se dotaz provedl a že trvá přes minutu.

Tato metoda dokáže ve své podstatě řešit i problém s načtením dat do paměti, který je zmiňován výše. Data je sice nezbytné načíst z disku do operační paměti databáze (pokud tam ještě nejsou), ale tento čas je dále rozmělněn mezi další (stejné) dotazy SQL, které následují hned po skončení toho prvního. Další dotazy již budou provedeny rychleji než ten první, který ještě čekal na natažení dat do paměti. Z toho tedy vyplývá, že se přiblížíme skutečné době vykonávání dotazu (pokud celou hodnotu vydělíme počtem zvládnutých dotazů) bez čekání na načtení dat z disku do paměti.

SQL_NO_CACHE – odpojení cache

Při testování výkonu dotazů je vhodné odpojit požadavky na jejich ukládání do mezipaměti (viz podkapitola Pokyny pro optimalizátor MySQL). Tento požadavek má pro testování zásadní význam. V situaci, kdy provádíme testy daných dotazů, požadujeme, aby se hodnota co nejvíce blížila reálné vykonávací době dotazu. Ačkoli nám jde o co nejlepší výsledek, tak tento výsledek požadujeme na základě vlastní efektivity dotazu. Nikoli na připojení vylepšujících mechanismů, jako je mezipaměť (angl. cache) databáze.

Necháme-li mezipaměť při testování vypnutou, získáme tedy reálnější představu o efektivitě, se kterou jsme dotaz SQL postavili. V reálném nasazení se samozřejmě tohoto vylepšení vzdávat nebudeme. Tam pochopitelně budou dotazy fungovat normálně spolu s mezipamětí. Nakonec tedy

můžeme očekávat i lepší výsledky výkonu dotazu. Pochopitelně za předpokladu, že se dotaz bude často opakovat a jeho místo ve velikostí omezené mezipaměti nebude nahrazeno.

3.3 Program QueryCounter

Na začátku této kapitoly jsem se zmínil o programu `QueryCouter`. Tento program mi posloužil k testování zkonstruovaných dotazů SQL a k měření jejich výkonu. Jedná se vůbec o první software, který v rámci tohoto projektu vznikl.

Na závěrech testů dotazů SQL, které byly provedeny programem `QueryCouter`, jsem později postavil celou aplikaci. Jelikož si tato podkapitola klade za úkol popsat především, jak tento software funguje a jaké jsou jeho možnosti, nebude zde ještě obsažena žádná z následně implementovaných myšlenek. Maximálně se zde mohou objevit některé konstrukce jazyka SQL, na kterých budu ukazovat možnosti testování.

3.3.1 Implementované testovací metody

Program `QueryCounter` umožňuje testování obou výše zmíněných metod. Testovaní dlouho trvajících dotazů (spočtení délky provádění jednoho dotazu) se zapíná přepínačem `t`, který daný dotaz zašle do databáze a po obdržení výsledku zapíše čas do výstupního souboru.

Zajímavostí na této metodě je skutečnost, že byla implementována až jako odpověď na potřebu testovat dlouho trvající dotazy. Tato potřeba se objevila v poměrně pokročilé fázi návrhu. Do té doby efektivita vykonávání navrhovaných konstrukcí SQL nevyžadovala testování na čas.

Samozřejmě program `QueryCounter` obsahuje i metodu QPM, kterou je možno spustit přepínačem `c`. Tato metoda opět své výsledky zaznamenává do výstupního souboru.

3.3.2 Možnosti vstupů

Vstup příkazovou řádkou

Program `QueryCounter` umožňuje testování dotazů přímo z příkazové řádky. Tím se mi splnila podmínka na testování elementárních dotazů, které nebylo třeba uchovávat v SQL skriptech. Dalo by se říci, že se jedná o jakousi okamžitou formu testu. Pochopitelně jsou opět možné obě metody (Time i QPM) a výsledek testů je zapsán do výstupního souboru.

Vstup textovým souborem

Druhou metodou je uchování konstrukcí SQL v souboru, jehož umístění se poskytuje programu (přepínač `f`). Tento soubor je v programu `QueryCounter` dále zpracováván. Tato metoda umožňuje testování mnohem komplexnějších konstrukcí SQL než při zaslání dotazu prostřednictvím příkazové řádky. Formát souboru pro tyto konstrukce SQL je popsán v následující podkapitole.

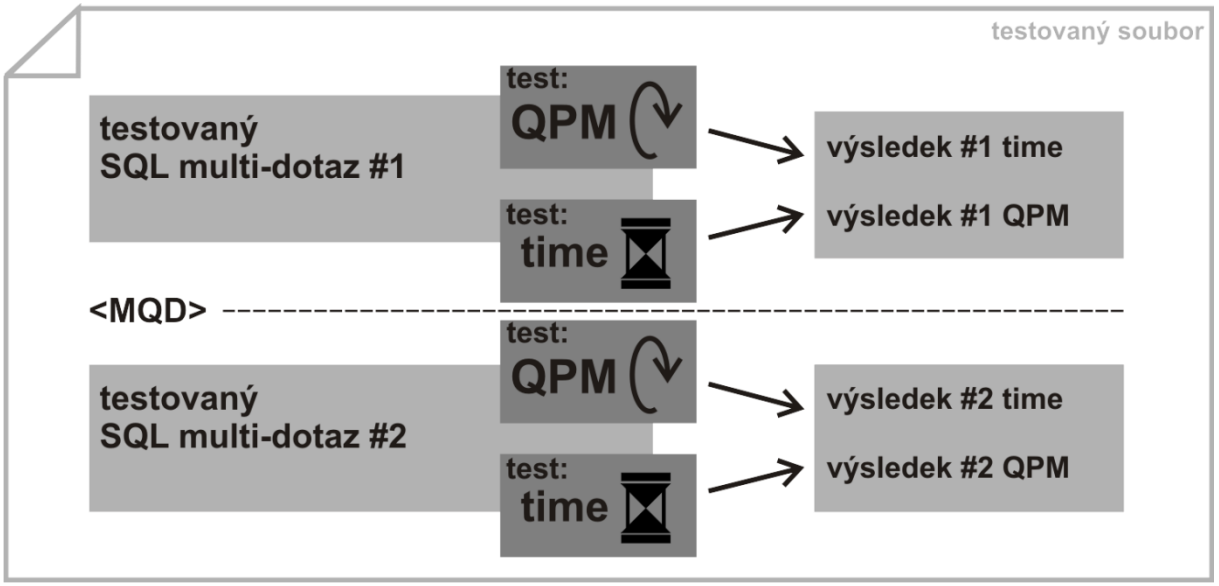
3.3.3 Formát textového souboru

Formát textového souboru se odvíjí od režimů, podle kterých chceme testovat. Tyto režimy jsou v programu `QueryCounter` definovány pomocí speciálních významových značek (dále jen tagů).

tag <MQD>

Tag `MQD` vychází z anglického sousloví Multi Query Delimiter. Tento tag v podstatě umožňuje rozdělit testovaný soubor SQL do několika na sobě nezávislých sekcí. Každá tato sekce je pak zpracovávána zvlášť. Tím je myšleno, že na každé sekci (konstrukci SQL), která je umístěna od začátku souboru po první tag `<MQD>`, je prováděno testování dle požadavků na vstupu (Time QPM).

To ve své podstatě umožňuje rozdělit testovaný soubor na jednotlivé sekce. Takže při testování nemusíme mít pro každý test specifický soubor (i to je ovšem možné), ale jsme schopni několik testů zapouzdřit do jediného souboru. Pro lepší pochopení tagu `<MQD>` je dále v práci umístěno grafické znázornění provádění takového souboru. Pokud se tag `<MQD>` nevyskytuje v daném testovaném souboru vůbec, pak je jeho obsah zpracováván jako jeden celek. A přitom nezávisí na tom, kolik je v něm dotazů.



Obrázek 3: Provádění souboru s <MQD> tagem

Z výše uvedených informací tedy vyplývá, že nejmenší zpracovávaná část testu (vstupního souboru) je jeden `<MQD>` blok, který obsahuje jeden až více dotazů SQL, které jsou databázi zasílány jako celek (v anglické terminologii označováno jako multi query).

tag <SINGLE>

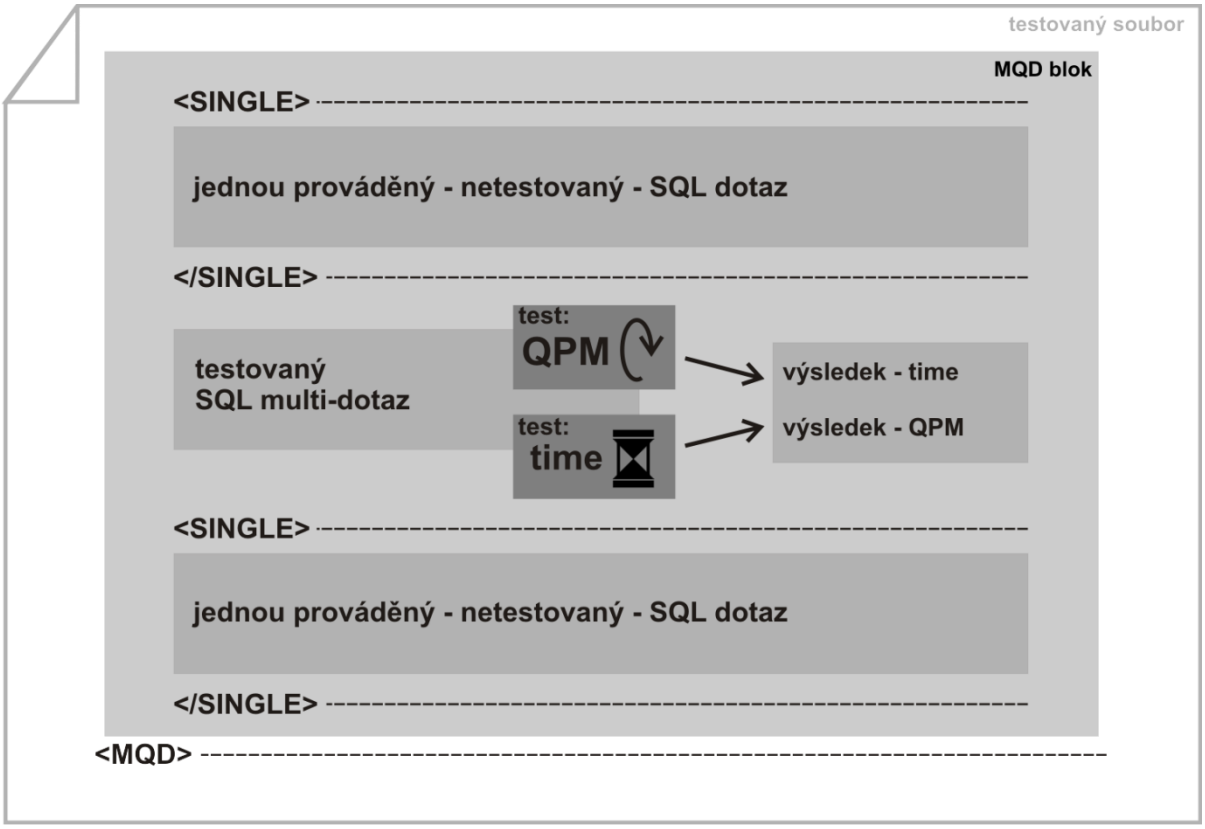
Během návrhu jsem se dostal až k potřebě, připravit si pro daný test jistou množinu dat. Tato data byla uložena v dočasné tabulce, kterou později navrhovaný algoritmus vyhledávání používá.

O dočasných tabulkách v databázovém stroji MySQL víme, že jejich použití je možné pouze v rámci jednoho připojení k databázi, pod kterým je temporární tabulka vytvořena. Jinými slovy, na dočasnou tabulku, která vznikla pod daným připojením, je vidět pouze z tohoto připojení a z žádného jiného.

Párový tag `<SINGLE>` (`</SINGLE>`) umožňuje spustit tu část dotazu SQL, kterou obaluje, jen jednou. To znamená, že daný dotaz se netestuje metodou QPM ani metodou Time, ale jen se provede, čímž může poskytnout právě tu výše zmíněnou datovou základnu pro další dotazy. Tohoto chování je dosaženo díky jednomu jedinému připojení k databázi. Připojení se nevytváří při každém provádění dotazu, takže v testu je možné použít i dočasné SQL tabulky.

Výskyt tohoto tagu není nijak omezen. Ale uživatel programu `QueryCounter` by měl pamatovat, že tento tag by měl vhodnou formou obalovat celek, který je později testován na výkon. Tento testovaný celek musí zůstat nerozdělen v rámci tagu `<MQD>`.

I přes relativně jasnou funkčnost tagu `<SINGLE>` uvádím níže v textu grafické znázornění popisující princip jeho provádění.



Obrázek 4: Provádění souboru s tagem <SINGLE>

3.3.4 Implementace a technické detaily programu QueryCounter

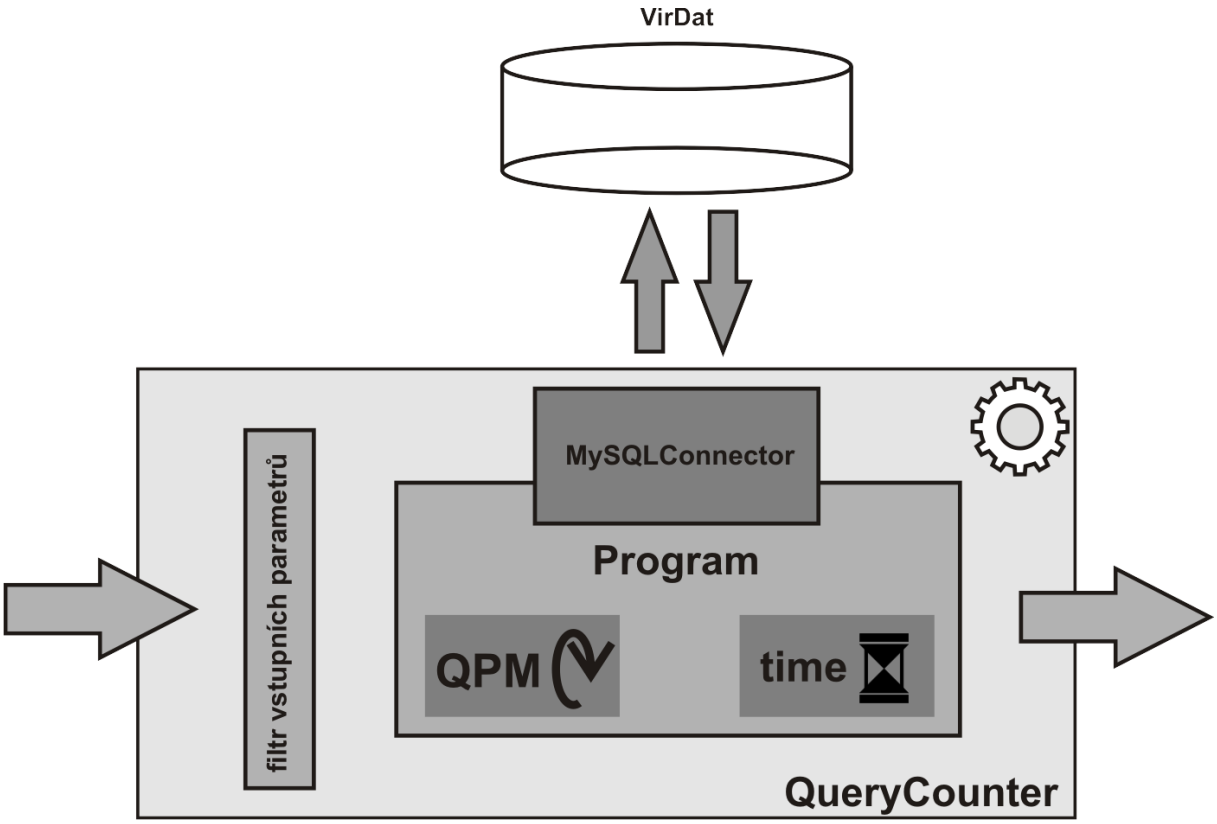
Celý nástroj je implementovaný v jazyce C# (stejně tak i později navrhovaný systém pro vyhledávání) a využívá konektoru k databázovému stroji MySQL, který je možné bezplatně stáhnout na [3].

Celý program je složen ze tří souborů zdrojových kódů.

První je soubor zdrojového kódu `MySQLConnector.cs`. Tento soubor implementuje mnou vytvořené uživatelské API nad databázovým konektorem ke stroji MySQL. V podstatě se jedná o sadu metod třídy `MySQLConnector`, která zjednodušuje a zaobaluje funkčnost výše zmíněného databázového konektoru a jeho metod.

Druhým souborem zdrojového kódu je `Program.cs` (a v něm obsažená třída `Program`), který nese logiku aplikace. Zde tedy vznikají požadavky na spojení s databází, rozebírání vstupních dat a řízení toku programu tak, aby splňoval požadavky na jednotlivé metody (QPM, Time).

Poslední souborem zdrojového kódu je `QueryCounter.cs`. Tento soubor vytváří pouze obálku nad výše popsanou logikou. Současně má na starosti převzetí a transformování vstupních parametrů a následnou kontrolu, zda jsou všechny nezbytné parametry zadány. Pokud jsou všechny požadavky splněny, spouští vlastní aplikaci.



Obrázek 5: Grafické znázornění vnitřní struktury programu `QueryCounter`

Z pohledu struktury se tedy jedná o tři do sebe zapouzdřené třídy. Program `QueryCounter` zasílá dotazy do databáze podle scénáře. Scénář je mu poskytnut buď ze vstupního souboru, nebo přímo dotazem a výstupem je soubor (angl. log), který obsahuje výsledky testů. Pro lepší pochopení struktury programu `QueryCounter` je výše přiloženo ještě jeho grafické znázornění včetně vstupů, komunikace a výstupů.

4 Návrh systému AVGMIS

Následující kapitola obsahuje zevrubný popis toho, jak jsem navrhoval nástroj pro vyhledávání informací nad databází `virdat`. Celá kapitola je rozdělena na dvě základní podkapitoly. První podkapitola se zabývá aplikační logikou, návrhem, tvorbou a testy jednotlivých dotazů.

Druhá kapitola se pak zabývá návrhem grafického uživatelského rozhraní, které celou aplikační logiku obaluje a vytváří tak pohodlnou možnost, jak tuto logiku využívat. Současně jsou v druhé kapitole zahrnuty i předlohy, ze kterých jsem vycházel na základě zadání společnosti AVG Technologies. Dále pak je daná podkapitola ještě rozložena na sekce vstup a výstup, kde je diskutována funkčnost vstupních a výstupního rozhraní vzhledem k požadavkům zadavatele.

4.1 Dotazy – jádro aplikační logiky

Celá tato podkapitola popisuje, jakým způsobem jsem navrhnul sestavování dotazů, v jakém pořadí jsou tyto dotazy vykonávány a jakých prostředků je využito pro skladování mezivýsledků. Dále pak zde najdete i formy dotazů SQL a důvody, proč jsou tyto konstrukce rychlejší než jiné testované.

Hned v následujících odstavcích naleznete popis jednotlivých atributů hlavně z pohledu možné četnosti výskytu v jednotlivých hledáních.

4.1.1 Požadované atributy a jejich možné četnosti

Atributy požadované zadáním jsou výše popsány v sekci, kde jsem diskutoval strukturu databáze `virdat` (kapitola Struktura databáze `virdat`). Zde jsem zanechal pouze informace o možných četnostech výskytů a specifických vlastnostech, které bylo pro jednotlivé atributy potřeba zohlednit. Je to z toho důvodu, že právě na těchto odstavcích je dobře pochopitelná komplexnost jednotlivých možných situací.

Po přečtení této sekce bude čtenář, doufám, souhlasit, že bylo nezbytné implementovat aplikační logiku do pokládání dotazů namísto tvorby jednoho komplexního a velmi neefektivního dotazu. V kapitole, která hodnotí úspěšnost provedených optimalizací, jsou uvedeny výsledky porovnání rychlosti neoptimalizovaných a optimalizovaných dotazů SQL.

Detekce

Požadavky na vyhledávání podle detekcí byly, co do rozměru požadavků zadavatele, nejkomplexnější v celé aplikaci. Prvním požadavkem bylo elementární vyhledávání podle jednotlivých detekcí a jejich antivirových tvůrců.

Jelikož se jedná o vyhledání textových řetězců, a v některých případech i poměrně dlouhých, bylo by velmi nekomfortní pro uživatele, aby musel zadávat detekční řetězec celý. Součástí

požadavků byla tedy i volba vyhledávání vloženého řetězce. Tento řetězec je tedy možno zadávat ve třech různých režimech:

- **přesná shoda** – zde se očekává, že vyhledávaný řetězec bude zadán v přesné formě, v jaké je uložen v databázi `virdat`.

- **otevřený konec** – jedná se o formu řetězce, který má shodný začátek a otevřený konec. Tedy zprava otevřený podřetězec.

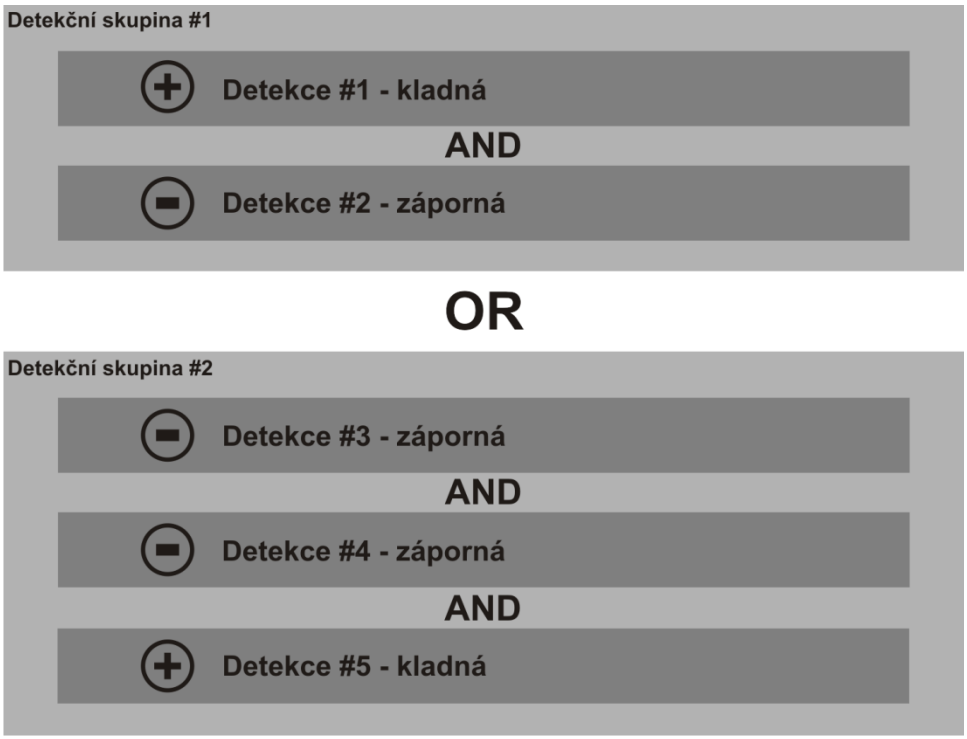
- **podřetězec** – benevolentnější předchozí forma. Oba konce jsou otevřené, takže se hledá pouze daný podřetězec. Tato operace je také vůbec nejdéle trvající formou vyhledávání.

Současně by vyhledávání podle detekcí mělo umožňovat, aby se ve hledaných vzorcích nevyskytovaly detekce určitého výrobce. Jinými slovy, hledáme takové vzorky, které daný výrobce nedetekuje (v tuto chvíli nezáleží na řetězci detekce). Z toho vyplývá, že se tu objevuje i hledání vzorků podle jejich záporných vlastností (hledáme vzorek, který něco nemá). To je ovšem v databázích SQL vždy náročnou operací, byť na to existuje klauzule SQL (`NOT EXISTS`).

Posledním požadavkem na vyhledávání podle detekcí bylo, aby se dala nasimulovat libovolná situace. Libovolnou situací je myšleno plné logické spektrum. Tedy možnost vyhledávat detekce formou různých průniků a sjednocení. Jako odpověď na tento požadavek jsem sestrojil kolekci `detectionGroup`. Jedná se o kolekci různých detekcí, které ale v dané skupině (kolekci) platí současně. V dané kolekci `detectionGroup` může být jedna až libovolné množství hledaných detekčních atributů.

Jelikož se zadání neomezuje na maximální počet detekcí, které vyhledáváme, může být i počet detekčních skupin relativně neomezený. Tyto detekční skupiny jsou mezi sebou ve vztahu OR. Tedy vybereme všechny vzorky, které platí pro skupinu jedna nebo pro skupinu dva. Opět se předpokládá, že v rámci vyhledávání podle detekcí bude alespoň jedna detekční skupina (`detectionGroup`).

Tento poněkud složitý slovní popis vysvětluje obrázek uvedený níže. Na obrázku je znázorněno několik detekčních skupin a uvnitř těchto skupin i několik detekcí (požadavků na ně). Jak je vidět z obrázku, jednotlivé detekce mohou být „kladné“ (požadavek na vzorek splňující tuto vlastnost), ale i „záporné“ (požadavek na vzorek, který požadovanou vlastnost nemá – není detekován daným výrobcem). Mezi jednotlivými požadavky jsou znázorněny logické značky, aby bylo názorně vidět, jaké platí mezi požadavky vztahy.



Obrázek 6: Grafické znázornění sady požadavků na vyhledávání podle detekcí

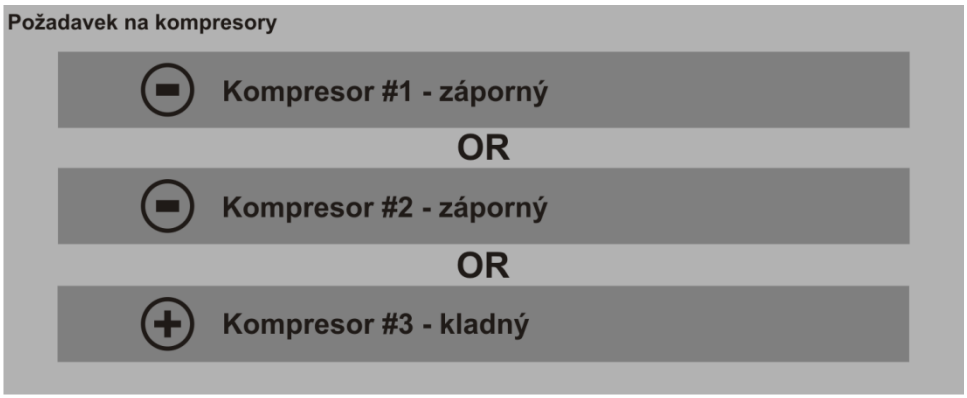
Archivátory, packery

Vyhledávání podle „archivátorů“ (tento výraz zde budu používat namísto výrazu archivačního softwaru) a „packerů“ (tento výraz zde budu používat namísto výrazu „runtime packer“) jsem zde spojil dohromady do jedné sekce. Principiálně se požadavky na vyhledávání těchto dvou atributů nijak neliší.

Pokud se podíváme výše do této práce, pak je v kapitole Struktura databáze `virdat` uvedena skutečnost, že jeden vzorek může být v jednom okamžiku zabalen pouze jedním „archivátorem“ či „packerem“. Ovšem neexistuje omezení, že jeden vzorek může být v různých výskytech zabalen různými kompresory (tento výraz zde budu dál používat namísto „archivátorů“ nebo „packerů“).

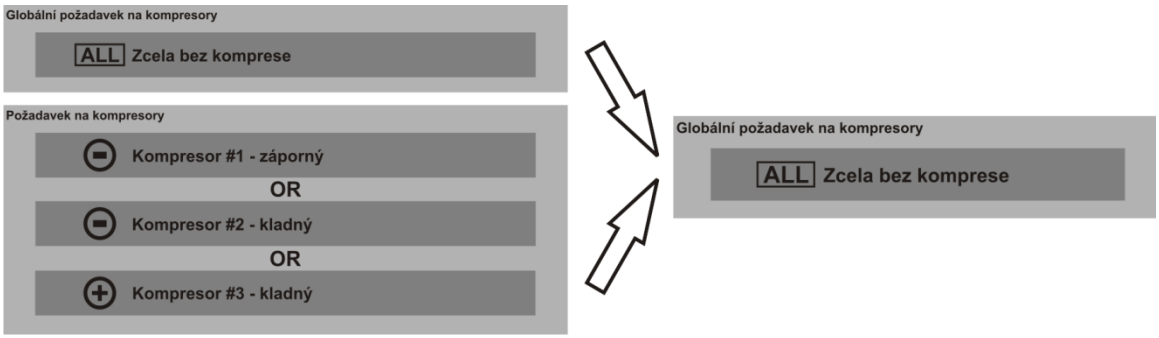
Z výše uvedeného odstavce tedy vyplývá, že daný vzorek v databázi (jeho dekomprimovaná podoba) může mít nula a více atributů v podobě různých kompresorů. Proto tedy vyhledávání podle jednotlivých kompresorů musí umožňovat zadat nulu až předem nedefinovaný počet těchto atributů. Současně tyto atributy jsou mezi sebou vzhledem k vyhledávanému vzorku v logické relaci `OR`. Opět zde uvádím grafické znázornění tohoto atributu níže v textu.

Dále pak bylo požadováno, aby systém umožnil vyhledávání záporných vlastností těchto atributů – tedy okamžik, kdy daný vzorek není zabalen určitých specifickým kompresorem (na rozdíl od detekcí zde již zadavatel požaduje, aby byl vložen konkrétní název kompresoru – zde by si měl čtenář uvědomit rozdíl v náročnosti vzhledem k databázi).



Obrázek 7: Grafické znázornění vztahů pro kompresory

Poslední požadavkem zadavatele byla situace, kdy daný vzorek není zabalen (ať už „archivátorem“ nebo „packerem“) vůbec. Pokud uživatel zvolí tuto variantu, pak se všechny ostatní požadavky z dané sekce kompresorů smažou a zůstane pouze tento „globální“.



Obrázek 8: Grafické znázornění vztahu mezi kompresory a globální požadavkem při současném užití

Skupiny

Skupiny jsou do jisté míry velmi podobné kompresorům. Jediným rozdílem je skutečnost, že daný vzorek se může nacházet ve dvou skupinách současně. Z toho vyplývá, že vztahy mezi jednotlivými skupinami nejsou typu `OR`, ale `AND`. Maximální počet zadávaných skupin by neměl být omezen, takže je zde opět možnost mít nula (skupiny uživatele nezajímají) až více požadavků na skupiny.

Zadavatel ovšem požadoval, stejně jako u kompresorů, aby se daly vyhledávat vzorky, které nejsou v některé z existujících skupin.

Informace o verzích

Tento překlad pochází z anglického sousloví „version info“. Jedná se o čtveřici textových atributů, které jsou umístěny v sekci `resources` hledaného vzorku. Popis těchto atributů je výše v kapitole Struktura databáze virdat.

Z pohledu návrhu aplikace se zde jednalo o vyhledávání řetězců umístěných v sloupcích tabulky `files`. Tyto sloupce bohužel nejsou indexovány a zadavatel požadoval, aby bylo možné

jejich hledání formou otevřeného konce (popis otevřeného konce je k dispozici v sekci Detekce v této podkapitole).

Specialitou na těchto atributech je skutečnost, kdy daný vzorek nemá žádné informace o verzích. Jelikož jsou tyto atributy zcela volitelné, pak tato situace není nijak neobvyklá. I když dobrý výrobce softwaru si svůj výrobek tímto způsobem rád popíše, jsou případy (a není jich málo), kdy tyto hodnoty prostě chybí. Z toho důvodu zadavatel požadoval vyhledávání takových vzorků, které tyto atributy nemají (v takové případě hledáme přímo hodnotu `NULL` nad danými sloupci).

Velikost

Celočíselný atribut, který má každý vzorek v databázi `virdat`. V tomto případě se tedy jedná o vyhledávání podle velikosti. Byť zadavatel blíže nespecifikoval vyhledávání podle velikosti, je celkem rozumné a uživatelsky příjemné umožnit vyhledávání v rámci různých intervalů. Současně jsem ponechal vyhledávání i podle přesné hodnoty. Tato volba je využívána při hledání některých velmi specifických vzorků, které mají vždy na bajt stejnou velikost.

Datum

Vyhledávání podle data je velmi podobné jako vyhledávání podle velikosti. Jediný přídavek ze strany zadavatele spočíval v několik „přednastavených“ hodnotách (den, týden a měsíc), které by hledání mělo umožňovat. Přednastavené hodnoty tedy mají umožňovat hledání vzorků, které dorazily za posledních dvacet čtyři hodin, sedm dní či jeden měsíc.

V případě data se jedná o vyhledávání nad sloupcem `FIRST_SEEN` databáze `virdat`. Tedy nad sloupcem, který nese informace o prvním výskytu daného vzorku v AVG (okamžik, kdy byl zařazen do databáze `virdat`).

Rozmanité atributy

Rozmanité atributy je souhrnný název pro skupinu vlastností, které se nedaly vložit do žádné výše zmíněné skupiny. Jedná se o tři hashe (`setchash`, `iconhash`, `SHA256` hash z certifikátu), typ vzorku (`MZPE`, `ELF`, `script`, …) a konečně o stav vzorku (`uknown`, `clean`, `infected`, …).

Vzhledem k četnosti jednotlivých atributů každý soubor může obsahovat nanejvýš jednu takovou hodnotu (soubory bez certifikátů a ikon a soubory, které nejsou typu portable executable, mají v databázi zaznamenán pouze typ a status).

Vzhledem k tvorbě dotazů SQL se jedná pouze o konstrukce, které vyhledávají jednu specifickou hodnotu (s předem danou velikostí – hashe, či kódy pro status a typ) na každý vyhledávaný atribut.

Limit

Posledním z požadavků zadavatele bylo omezení výsledné množiny navrácených vzorků. Aplikace by při nevhodně zvolených hodnotách vyhledávaných atributů neměla být schopna vrátit celou databázi.

Tato sekce popisuje problematiku, jež spočívá v nastavení limitu pro výslednou sadu vzorků. Limit by měl být takový, aby tato sada byla dostatečně velká pro pozdější analytickou činnost. Vzpomeňte si na jednu ze zásad týkající se limitu a popsanou v sekci Omezení počtu řádků, která říká „nestahujte více dat, než skutečně využijete“.

Zde bych uvedl ještě příklad využití, na kterém se omezení výsledku pěkně ukáže. Virový analytik má před sebou neznámý vzorek. Z tohoto vzorku se následující analýzou zjistí několik jeho vlastností. Tyto vlastnosti jsou našimi vyhledávacími atributy. Aplikace `AVGMIS` mu po zadání atributů nabídne několik podobných vzorků (pouze na základě hledaných atributů). Dále pak jsou některé z nalezených vzorků uloženy v datových úložištích. Pokud si tyto vzorky prostřednictvím systému `AVGMIS` (či jiné používané aplikace) stáhne, může provádět jejich bližší porovnání. Nyní jsme u podstaty věci. Analytikovi by měl k jeho bližší analýze postačit jen zlomek (v řádu desítek až stovek kusů) z potenciálně obrovského množství nalezených vzorků. Proto stahovat celou sadu může být (a obvykle bývá) zbytečné.

4.1.2 Řízení vyhledávání – tabulka ctrl_avgmis

Po definici požadavků na atributy bylo jasně specifikováno, jaké tabulky budou požívány a jaké sloupce z těchto tabulek bude nezbytné využívat, aby byl systém schopen vrátit požadovanou množinu vzorků. Další částí návrhu tedy byla potřeba vytvořit takový mechanismus, který by automaticky řídil vyhledávání.

Pod pojmem řízení vyhledávání je myšlen algoritmus využívající takovou množinu dat, která obsahuje informace právě o výše zmíněných názvech tabulek a názvech sloupců. Součástí této množiny dat by měla být i informace o počtu vzorků, které mohou být vyhledávacími atributy ovlivněny. To jsou takové vzorky, které mají danou vlastnost (vyhledávaný atribut) a budou tudíž brány v potaz při vyhledávání.

Pro uložení hodnot zmíněných v předchozím odstavci byla vytvořena tabulka `ctrl_avgmis` (zkratka sousloví „control AVGMIS“), která je umístěna v rámci databáze `virdat`. Její obsah je uveden v přílohách na konci práce (konkrétně jako příloha tabulka `ctrl_avgmis`).

Ve finální verzi této řídicí tabulky (během postupného vývoje byly sloupce přidávány) jsou obsaženy následující informace:

- **název atributu** – označený jako „`section`“.
- **typ indexu** – „`index_type`“ – je označení prováděcí skupiny – tento sloupec a celé prováděcí pozadí bude uvedeno dále v této kapitole.

- **vztah mezi hodnotami** – „`relation`“ – nabývá hodnot nula nebo jedna v závislosti na skutečnosti, jestli daný atribut může nabývat více hodnot současně.
- **informace pro spojení** – „`r_table`“ a „`r_join_attrib`“ – hodnoty popisující název tabulky pro spojení a název konkrétního sloupce.
- **informace pro umístění hodnoty** – „`c_table`“ a „`c_join_attrib`“ – hodnoty popisující názvy tabulek a sloupců, které jsou použity na připojení slovníku (slovník je nositelem vyhledávané hodnoty).
- **sloupec s podmínkou** – „`c_where`“ – nese názvy sloupců v tabulkách. Tyto sloupce jsou použity v podmínkových částech (za klauzulí `WHERE`) v dotazech.

section	index_type	relation	rec_count	r_table	r_join_attrib	c_table	c_join_attrib	c_where
certificate	0	0	555892	r_files_certificates	id_certificate	c_certificates	id_certificate	SHA256
group	0	1	1919625	r_files_groups	id_group	c_groups	id_group	id_group
archiver	0	1	3615944	r_file_archiver	id_archiver	c_archivers	id_archiver	archiver
det_ms	1	0	3978745	r_files_det_ms	id_det_ms	c_det_ms	id_det_ms	det_ms
filedesc	3	0	6993719	NULL	NULL	files	NULL	VERINFO_FileDescription

Obrázek 9: Grafické znázornění několika řádků tabulky ctrl_avgmis

Hodnoty ve sloupcích „`r_table`“, „`r_join_attrib`“ a „`c_join_attrib`“ mohou být `null` (tedy nevyplněné). Bývá to v těch případech, kdy jsou slovníkové hodnoty umístěny přímo do bázové tabulky `files`, a tudíž k nim neexistuje žádná vazební tabulka ani slovník.

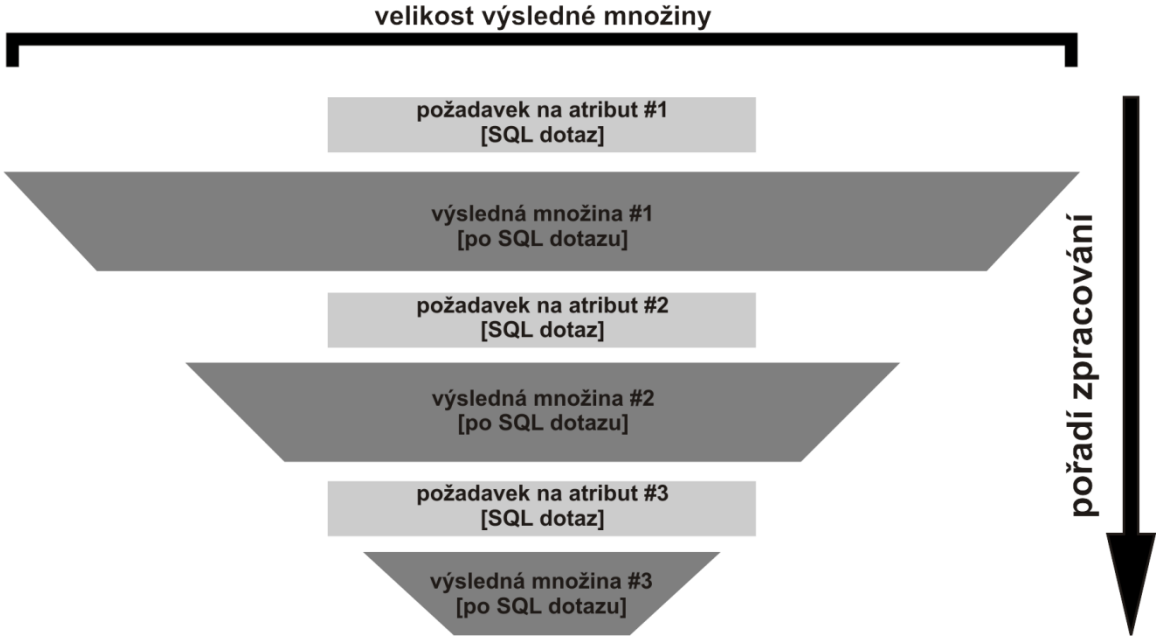
4.1.3 Algoritmus řídící vyhledávání

Algoritmus, který řídí vyhledávání, prošel od samého počátky vývoje nástroje několika signifikantními změnami. V tomto algoritmu jde zejména o vhodné sestavení pořadí vyhledávání jednotlivých hledaných atributů.

Pořadí, v jakém jsou jednotlivé atributy vyhledávány, je dáno počtem řádků, které daný atribut ovlivňuje (informace „`rec_count`“ obsažena v tabulce `ctrl_avgmis`). Čím méně ovlivněných vzorků daný atribut má, tím výše ve výsledném dotazu SQL bude uveden. Tohle je obecná myšlenka řízení pořadí. V následujících odstavcích je tato myšlenka rozvedena a následně zakomponována do vyhledávací strategie.

V prvotní verzi algoritmu vytvářejícího pořadí vyhledávání se počítalo s postupným vykonáváním jednotlivých požadovaných atributů (každý požadovaný atribut = jeden dotaz SQL na daný atribut). Výsledky hledání by se umísťovaly do výsledné dočasné tabulky a v každém dalším zpracování by se provádělo její spojení s dalším požadavkem. Podstata myšlenky spočívala ve skutečnosti, že první dotaz na atribut s nejmenší počtem ovlivněných záznamů (viz pořadí popsané výše) vytvoří také nejmenší množinu vrácených výsledků. Tím pádem i nejmenší možnou množinu

pro další spojení (při hledání dalšího atributu). Tento přístup byl ovšem časem označen jako vysoce neefektivní (zvláště pokud uživatel požadoval pouze atributy s vysokým výskytem – postupné zpracování trvalo déle než sloučení atributů do jednoho dotazu). Pro lepší představu je níže uvedeno grafické znázornění takového zpracování.



Obrázek 10: Grafické znázornění zpracování původní verzí algoritmu – pro tři hledané atributy

Aktuální a později implementovaná metoda si z té původní vzala pouze myšlenku pořadí skladby dotazu pro jednotlivé atributy. Aby byl omezen počet jednotlivých dotazů (jeden dotaz na každý atribut – kladný i záporný), byla vymyšlena nová verze tohoto algoritmu. Tato verze využívá speciálně definovaných skupin, které umožňují zmenšit počet dotazů do databáze. Tento mechanismus a popis jednotlivých skupin je vysvětlen hned v následující sekci této kapitoly. Výsledky měly být v původním znění ukládány do dočasných tabulek (jedna tabulka pro každou skupinu) a ty měly být pak na konci vyhledávání spojovány do výsledné množiny, která byla vrácena jako finální výsledek. Ovšem ani tento princip chování výsledků nebyl shledán dostatečně účinným, a proto vzniklo později kaskádové zpracování. Tento finální princip je uveden v sekci nazvané „Kaskádové zpracování“ níže v diplomové práci.

Čtenář může být po přečtení předchozích odstavců trochu zmatený, neboť se zde prolínají tři hlavní vývojové scénáře. Proto bych zde uvedl, jak funguje algoritmus řízení vyhledávání:

Vstup: požadavky uživatele ve formě hodnot získaných z webového rozhraní.
Výstup: seznam identifikátorů vzorků, které odpovídají daným hledaným atributům.
Metoda:

1. Požadované atributy jsou rozděleny do skupin na základě informací ze vstupu a tabulky ctrl_avgmis.
2. Z požadovaných skupin jsou sestaveny dotazy SQL (dotazy obsahují několik podmínek – dle požadovaných atributů) vzhledem k povaze každého skupinového vyhledávacího scénáře (pozitivní, smíšený, negativní – bude popsáno dále ve vyhledávacích skupinách).
3. Dotazy jsou kaskádovitě zpracovány s možností zkráceného vyhodnocení v případě potřeby.
4. Výsledná tabulka identifikátorů (poslední dočasná tabulka kaskády) je vrácena.

Algoritmus řízení vyhledávání

4.1.3.1 Vyhledávací skupiny a jejich bázové tabulky

Každý atribut, který systém AVGMIS umožňuje vyhledávat, spadá do některé ze tří vyhledávacích skupin. Každá z těchto skupin využívá jednoho specifického způsobu (algoritmu) pro tvorbu dotazů SQL. Tyto dotazy pak využívají zásady optimalizací dotazů, které jsou popsány v druhé kapitole této práce. Současně každá z těchto skupin může obsahovat různé scénáře vyhledávání na základě povah jednotlivých atributů (viz sekce Požadované atributy a jejich možné četnosti).

Názvy vyhledávacích skupin:

- skupina strukturálních indexů,
- skupina detekcí,
- skupina files.

Scénářů vyhledávání:

- **Pozitivní** – jedná se o situaci, kdy jsou požadovány vzorky, které všechny mají danou vlastnost (atribut). To v přeneseném slova smyslu v jazyce SQL znamená, že dotaz obsahuje pouze podmínky typu „rovno hodnotě“.
- **Negativní** – toto je situace, kdy požadujeme takové vzorky, které všechny nemají danou skupinu hledaných atributů. Pokud se na to podíváme opět z pohledu jazyka SQL, znamená to, že hledáme vzorky, obsahují podmínky „je různý od“ nebo NOT EXISTS.

- **Smíšený** – tato varianta je kombinací předchozích dvou scénářů. To znamená, že daný vzorek má určité atributy, ale současně jsou na něm požadovány i atributy, které mu chybí. V jazyce SQL je to kombinace „rovná se“ i „je různé od“.

Podstata existence jednotlivých scénářů spočívá ve skutečnosti, že vyhledávání záporu se musí řešit jinak než jen pouhou adekvátní konstrukcí SQL. Takové konstrukce obvykle nejsou efektivní a byly by v rozporu s účelem této práce.

Následující sekce obsahují kromě informací k daným skupinám i návrhy na zpracovávání jednotlivých scénářů. Informace o návrzích (ve formě diagramů) zpracovávání jednotlivých scénářů obsahují i části různých optimalizací (pokud jsou možné – zejména v části skupiny detekcí). Tyto optimalizace vedou zejména k minimalizaci počtu dotazů při zachování jejich logické správnosti. Tyto návrhy jsou uvedeny vždy na konci sekce.

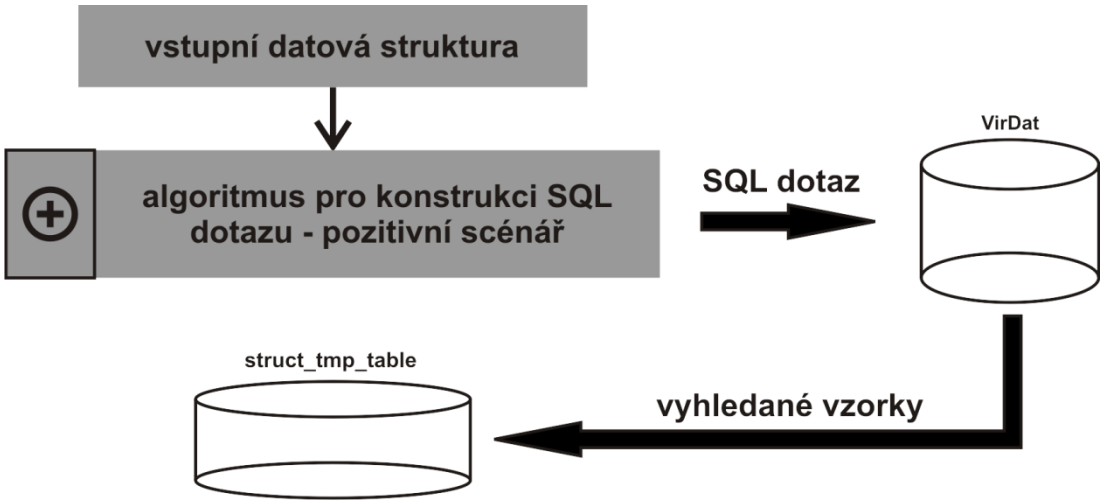
4.1.3.2 Skupina strukturálních indexů

Do této skupiny spadají všechny atributy, které ve sloupci `index_type` v tabulce `ctrl_avgmis` obsahují hodnotu nula. Název skupiny vznikl na základě podobností struktur SQL (tabulek a sloupců), ve kterých jsou atributy uloženy. Každý atribut má svou vazební tabulku (`r_file(s)_???`) a v ní podobně nebo stejně indexované sloupce (záleží jen na četnosti výskytů). Hodnoty jsou uloženy ve slovnících (`c_???`) s podobnou či stejnou strukturou. Proto je možné tuto skupinu cyklicky programově zpracovávat a postupně tak vytvářet dotaz SQL. O tom, jak vypadají konkrétní dotazy, je možné se dočíst v následujících sekcích.

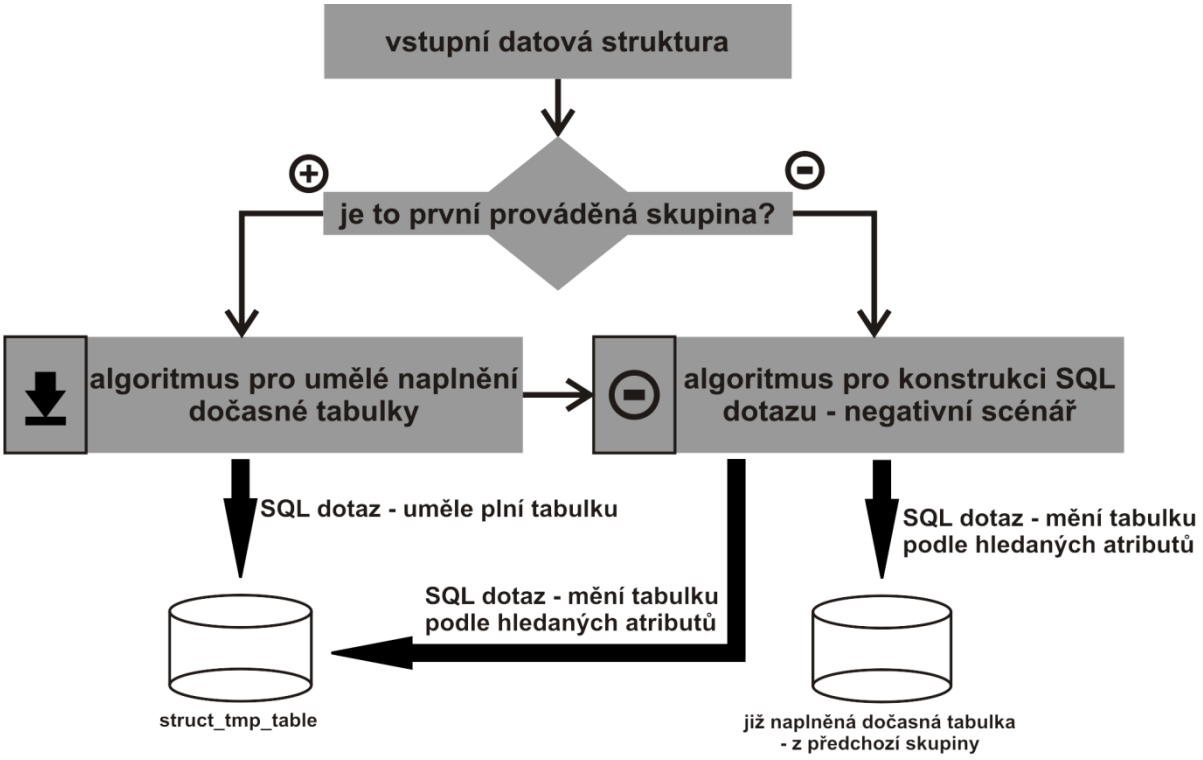
Strukturální skupina může používat všech tří výše pospaných scénářů. Je to jednoduše z toho důvodu, že atributy zahrnuté do této skupiny zpracování mohou nabývat jak pozitivních (vzorek obsahuje tuto vlastnost), tak negativních hodnot (vzorek neobsahuje tuto vlastnost). Každý scénář tedy později funguje na jiném principu a generuje typově jiný dotaz SQL.

Získaná data jsou posléze ukládána do dočasné tabulky, která se jmenuje „`struct_tmp_table`“.

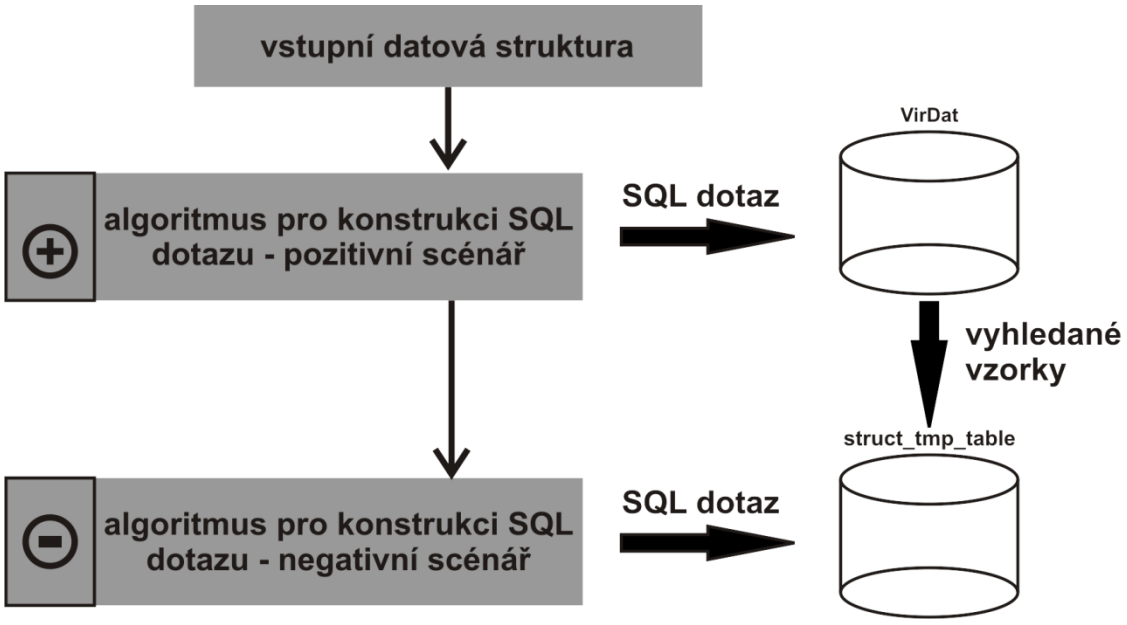
Návrhy zpracování



Obrázek 11: Diagram znázorňující návrh zpracování skupiny strukturálních indexů při pozitivním scénáři



Obrázek 12: Diagram znázorňující návrh zpracování skupiny strukturálních indexů při negativním scénáři



Obrázek 13: Diagram znázorňující návrh zpracování skupiny strukturálních indexů při smíšeném scénáři

4.1.3.3 Skupina files

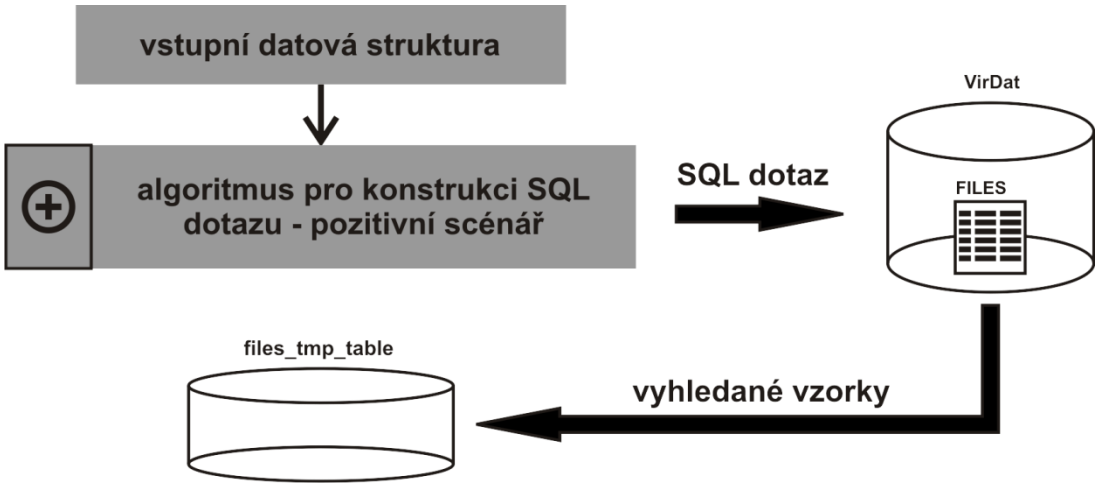
Další specifickou skupinou je skupina `files`. Atributy náležící do této skupiny jsou specifické tím, že jsou všechny umístěny v tabulce `files`. Tedy jediná struktura, která je procházena kvůli nalezení požadovaných vzorků, je tabulka `files`. Důvod, proč jsou všechny tyto atributy zpracovávány v rámci jedné skupiny, je skutečnost, že není potřeba připojovat žádnou další tabulku k této obrovské bázové tabulce. Hodnoty sloupce „`index_type`“ pro tuto skupinu jsou dvě a tři.

Důvod, proč u skupiny `files` nestačí jen jedno číslo, je ten, že i atributy jsou v rámci tabulky `files` různé. Hodnota dvě popisuje atributy, které jsou v rámci tabulky indexovány. To zaručuje při správném použití (vhodně zvolené pořadí v podmínkové části dotazu SQL) vyšší výkon dotazu. Atributy s hodnotou tři pak nejsou indexovány vůbec, ale jsou v některých případech vhodným adeptem k indexování (více o těchto vylepšeních dále v práci). Atributy, které nejsou potaženy žádným indexem, v tabulce `files` reprezentují obecně nejhorší možnou verzi hledání (další informace jsou uvedeny dále v sekci Kaskádového zpracování).

Co se scénářů pro tuto skupinu týče, tak vzhledem k požadavkům zadavatele je možná jen pozitivní varianta hledání. To celou situaci při tvorbě dotazů SQL výrazně zjednodušuje. Ostatně vzhledem k neexistenci indexace nad některými atributy této skupiny zde není příliš prostoru pro využití některých výše zmíněných rad.

Získaná data jsou i zde ukládána do výsledné dočasné tabulky. Její název je „`files_tmp_table`“. Obvykle právě tato tabulka funguje jako finální zdroj identifikátorů, protože právě absence indexace ji odsouvá až na poslední místo kaskádového zpracování.

Návrhy zpracování



Obrázek 14: Diagram znázorňující návrh zpracování skupiny files (možný pouze pozitivní scénář)

4.1.3.4 Skupina detekcí

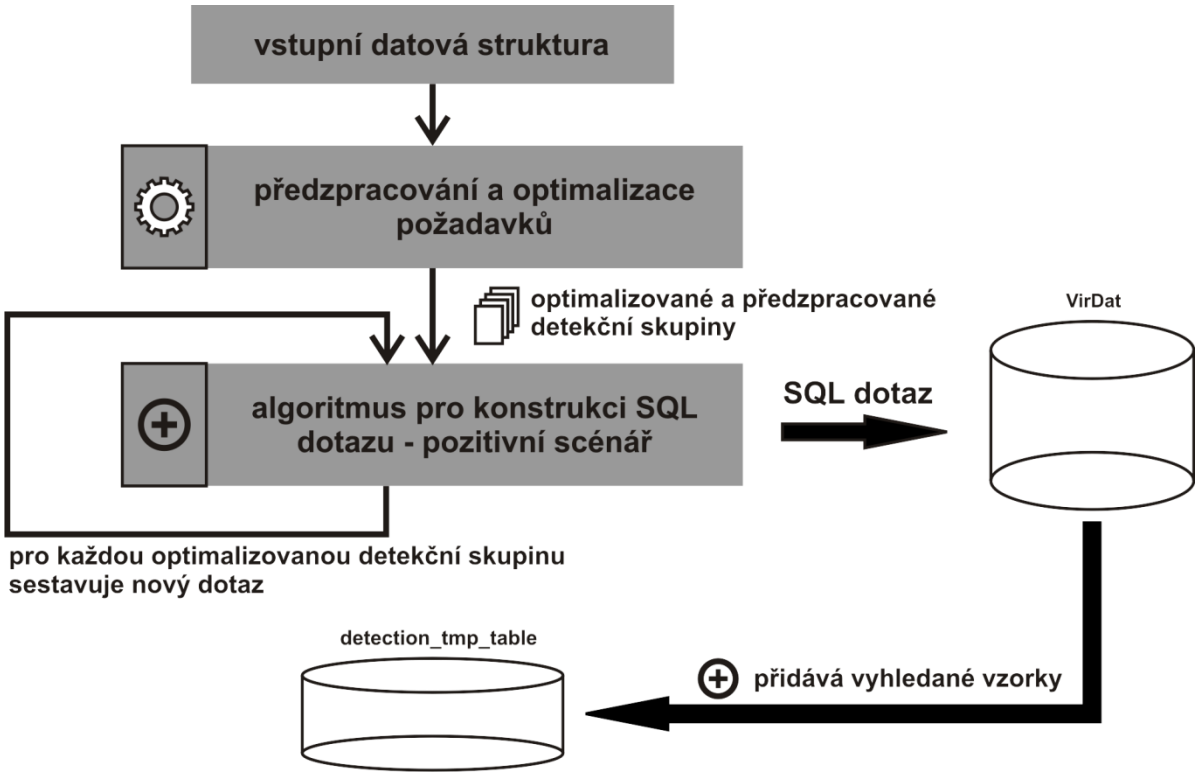
Tato skupina byla definována jako samostatná ze dvou důvodů. Prvním a původním důvodem byla historická skutečnost, že pro zpracování dotazů na detekce byla vytvořena vestavěná metoda. Tato metoda (konkrétně `sp_detection`) nebyla přímo navržena pro tento nástroj, a proto se později její použití stalo v podstatě nemyslitelným. Bohužel její efektivita nebyla dostatečná a postupem času byla vytvořena jiná, výkonnější varianta, která je implementována v aplikaci `AVGMIS`.

Druhý důvod, proč je tato skupina samostatná, vychází z požadavků zadavatele na možnosti detekcí. Pokud si čtenář přečte znovu požadavky na hledání podle detekcí, zjistí, že výsledkem může být i několik vzájemně oddělených výsledných množin. Všechny tyto varianty se odvíjí od složitosti, s jakou je vložen uživatelský požadavek na hledání. Ještě zde stojí za zmínku, že tato skupina může být tak komplexní, že předtím, než je provedeno sestavení dotazu, je potřeba zkusit aplikovat optimalizaci požadavků. Z toho důvodu bylo potřeba navrhnout optimalizátor požadovaných detekcí. Optimalizaci této skupiny je věnováno několik odstavců v následující sekci.

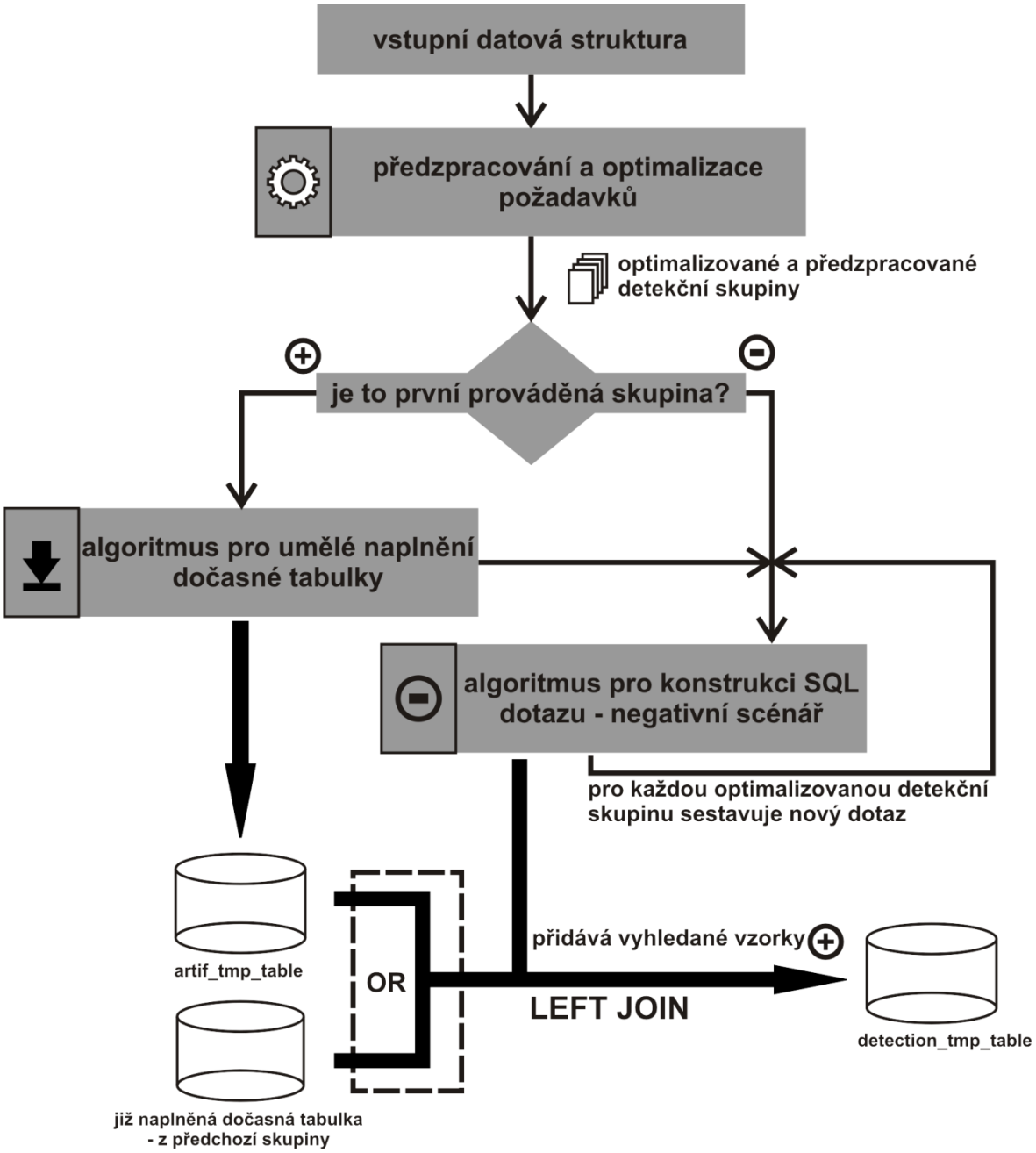
Ve sloupci `index_type` tabulky `ctrl_avgmis` zastávají atributy použité pro detekce hodnotu jedna. Výsledná data jsou ukládána do dočasné tabulky `detection_tmp_table`. Povaha hledání podle detekcí ovšem vyžadovala další tabulku pro odkládání některých potenciálně znovupoužitelných dat. Z toho důvodu existuje pro toto úložiště dočasná tabulka `temp_detection_tmp_table`.

Z pohledu možných scénářů nemá vyhledávání podle detekcí žádná omezení, takže jsou umožněny pozitivní, negativní a samozřejmě i smíšené scénáře.

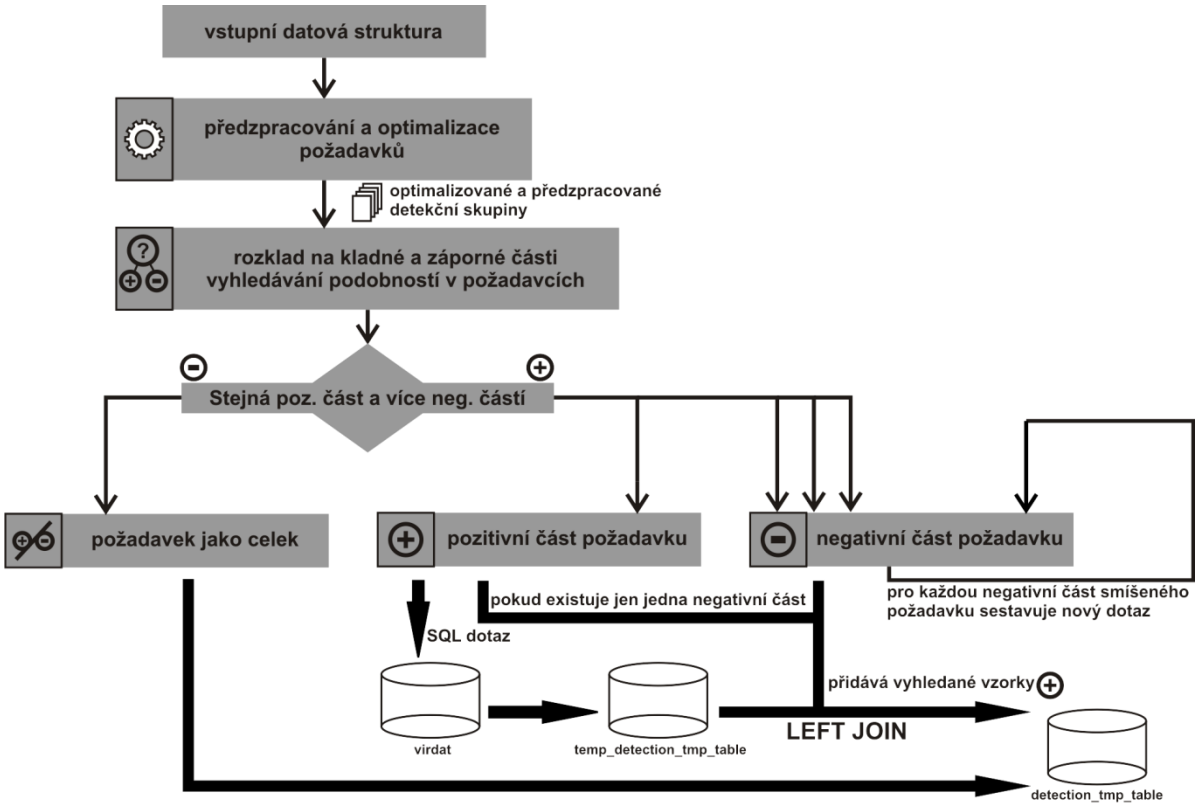
Návrhy zpracování



Obrázek 15: Diagram znázorňující návrh zpracování skupiny detekcí při pozitivním scénáři



Obrázek 16: Diagram znázorňující návrh zpracování skupiny detekcí při negativním scénáři

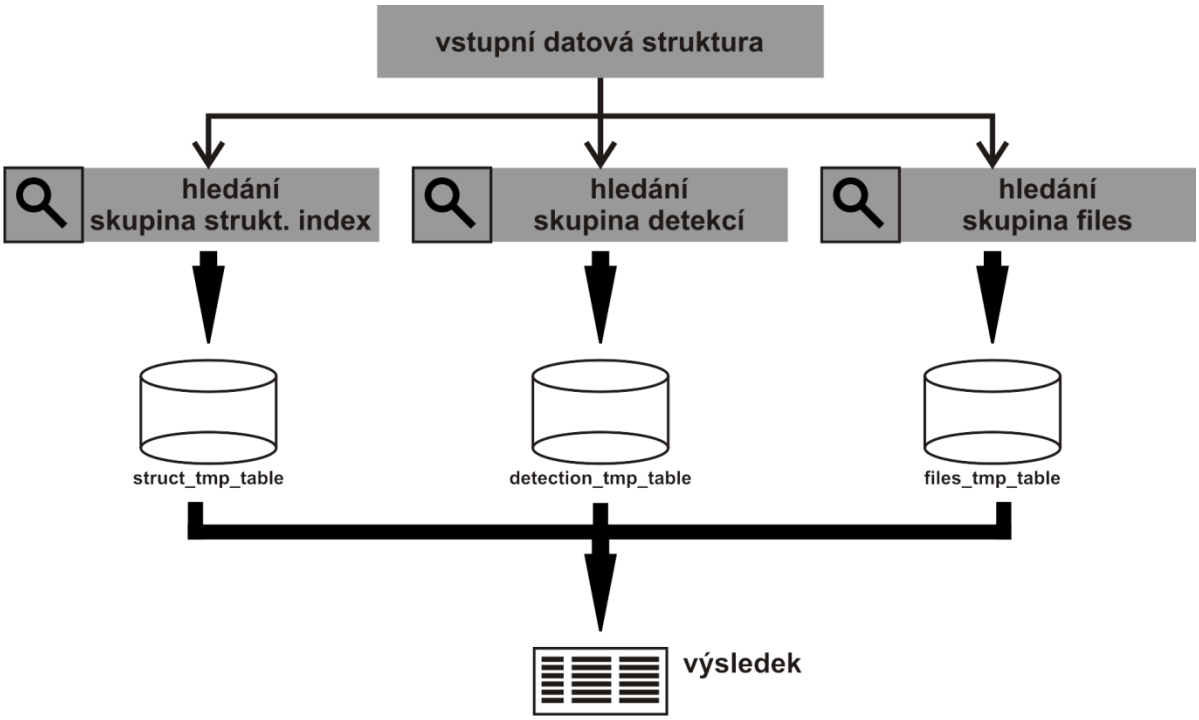


Obrázek 17: Diagram znázorňující návrh zpracování skupiny detekcí při smíšeném scénáři

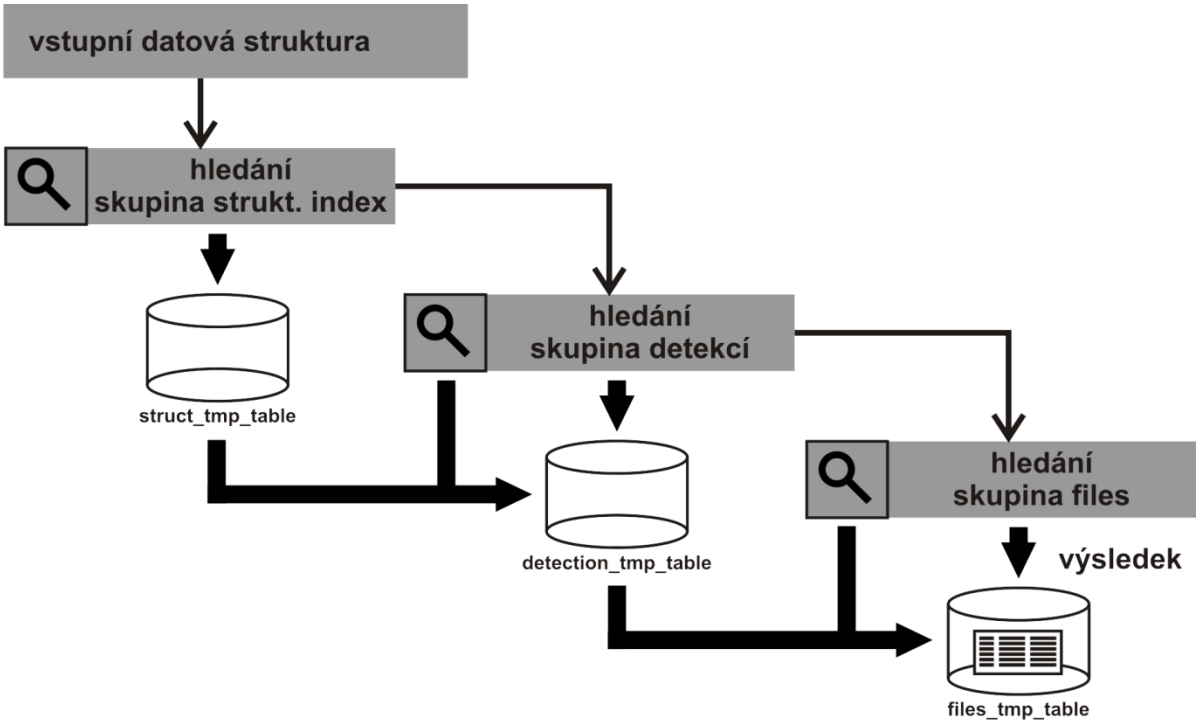
4.1.4 Kaskádové zpracování a hierarchie scénářů

Původní návrh zpracování počítal s odděleným prováděním jednotlivých skupin. Výsledná množina identifikátorů vzorků by poté vznikla vnitřním spojením jednotlivých výsledných dočasných tabulek. Grafické znázornění takového návrhu je uvedeno níže v textu.

Důvod, proč tento přístup byl nakonec zatracen a byl navržen nový, spočíval ve snaze využít nalezené vzorky z předchozího hledání (zpracování předchozí skupiny). Návrh kaskádového zpracování skupin říká, že data, která již byla nalezena v předchozím zpracování, by měla být využita při zpracování další skupiny. Data jsou využita ve formě spojení dočasné tabulky, která je výsledkem předchozího kroku a vytvářené konstrukce z následující zpracovávané skupiny. Tento princip umožní omezit v první řadě počet řádků během spojení v dalším dotazu (výsledný počet vzorků ve skupině nebude vyšší, než je počet řádků v dočasné tabulce). Aby byl tento přístup zcela jasný, je níže v textu uvedeno i grafické znázornění kaskádového zpracování. Jedná se o nejobecnější verzi kaskády, při které nejsou řešeny povahy scénářů (všechny scénáře jsou pozitivní a hledají se atributy všech tří skupin).



Obrázek 18: Grafické znázornění zpracování bez kaskádového efektu



Obrázek 19: Grafické znázornění zpracovávání s kaskádovým efektem

Dalším pozitivem této metody je možné zkrácené vyhodnocení, které je popsáno v následující sekci.

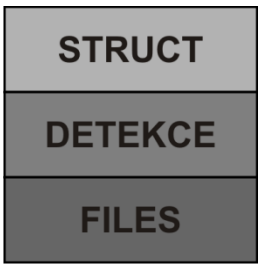
Po přečtení výše zmíněných principů kaskádového zpracování možná některé čtenáře napadne, v jakém pořadí se mají skupiny zpracovávat a záleží-li vůbec na pořadí zpracování skupin. Odpověď je ano, na pořadí zpracování skupin záleží a současně záleží na scénáři dané skupiny.

Na celou problematiku je potřeba nahlížet ze dvou stran. První sledovaný aspekt jsou skupiny a jejich rychlost zpracování a druhý je typ scénáře v dané skupině.

Pokud začneme typy skupin, pak je dáno, že některé skupiny mohou být obecně rychlejší než jiné. Skupina strukturálních indexů, tak jak je navržena, vyhledává data z oblastí, které jsou indexovány. Jejich nalezení bude tedy trvat nejkratší dobu.

Druhým v pořadí jsou detekce. Sice i jejich struktura tabulek je založena na indexaci podobně jako u skupiny strukturálních indexů, ale je tu jeden aspekt, který může celou situaci velmi zpomalit. Tím aspektem jsou detekční skupiny. Vzhledem k tomu, že jejich počet není zcela omezen a mají mezi sebou logickou vazbu OR, nelze je zpracovávat v jednom jednoduchém dotazu s vhodně nastavenými podmínkami – podobně jako skupina strukturálních indexů. Jejich zpracovávání tedy může trvat o dost delší dobu než zpracování předchozí skupiny.

Poslední skupinou jsou atributy s indexním typem dva a tři. V tomto případě se může zpracování protáhnout na výrazně delší dobu, jelikož mohou být požadovány pouze atributy, které nejsou pokryté indexem. V takové případě bude databázový stroj sekvenčně procházet přes čtyřicet milionů záznamů (k datu 24. 11. 2011). Opět zde následuje grafické znázornění hierarchie pro vyhledávací skupiny.



Obrázek 20: Grafické znázornění hierarchie skupin

Nyní, když existuje návrh na hierarchické zpracování skupin, je třeba ještě do něj vložit typy scénářů, které mohou dané skupiny nabízet. Obecně máme scénáře tři.

Prvním, a hierarchicky nejvyšším, scénářem bude scénář pozitivní. Je to z toho důvodu, že tento scénář je schopen generovat prvotní data. To znamená, že když celé vyhledávání začne, neexistuje žádná dočasná tabulka, se kterou bychom prováděli spojení. Pozitivní scénář nám je tedy schopen vytvořit základní množinu dat, se kterou můžeme spojovat další scénáře dalších skupin.

Druhý scénářem v hierarchii bude scénář smíšeného vyhledávání. Podstata opět spočívá v tom, jestli jsme schopni takovým požadavkem vygenerovat alespoň nějaká data, která budeme později spojovat v kaskádě. Zde je odpověď kladná, protože smíšený scénář má pozitivní část a od této části jsme již schopni „odečíst tu zápornou část“.

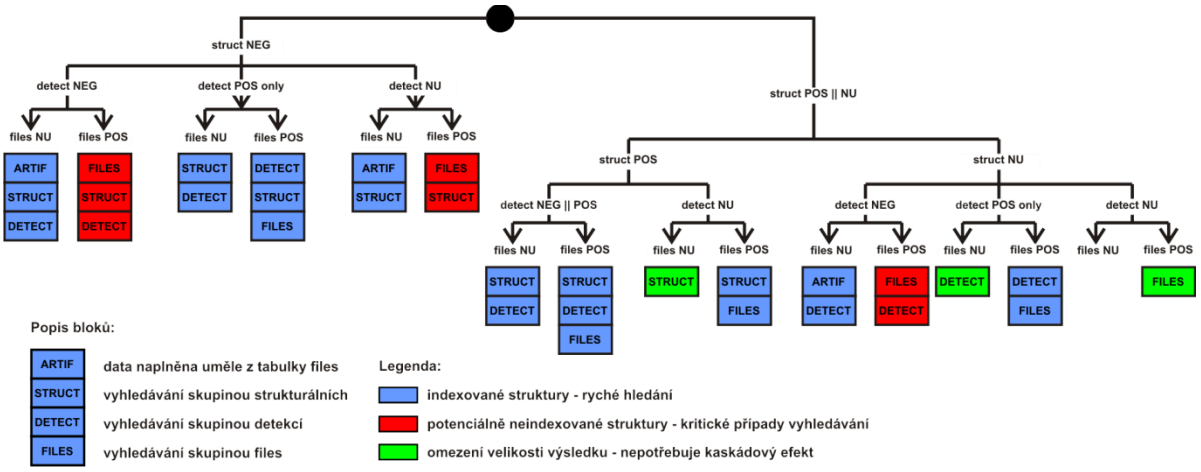
Posledním článkem hierarchie je scénář negativní. Takový scénář chceme provádět až v okamžiku, když už můžeme od něčeho odečítat. Proto se jej budeme snažit vykonat až nakonec. V případě, že negativní scénář některé skupiny (pokud ho skupina vůbec umožňuje) je jediný

prováděný, pak je třeba vytvořit tzv. „umělou sadu“ (nebo též „umělé plnění“), od které budeme odečítat.

Obrázek 21: Grafické znázornění hierarchie scénářů

Nyní byly definovány dva aspekty, které je třeba brát v potaz při sestavování hierarchie vyhledávání. Nyní oba dva spojíme do jednoho koncového návrhu posloupnosti vyhledávání. Zde je důležité pochopit, co je silnější argument (skupina, nebo scénář skupiny) a čím je tedy hierarchie vyhledávání řízena. Silnější argumentem je scénář. Tato skutečnost vychází z nutnosti mít nějaká data, ze kterých se dá později odečítat (v případě potřeby). Sice v jazyce SQL existují konstrukce, které dovedou vyhledat záznamy, které nemají danou vlastnost, a tím bychom se vyhnuli potřebě hledání pozitivních scénářů, ale jejich efektivita není dost vysoká. Obecně je tedy skupina až druhým argumentem pro sestavení hierarchie vyhledávání. Tuto negativní vlastnost se návrh snaží kompenzovat pokud možno co nejvíce efektivními dotazy SQL.

Návrh systému tedy počítá s celkem s šestnácti různými variantami hledání. Tyto varianty jsou později vyobrazeny v grafickém znázornění níže v textu. Ovšem i přes snahu učinit tento návrh co možná nejefektivnějším existují celkem tři varianty, kdy se bude muset potenciálně projít celá tabulka `files`. V případě, že budeme hledat neindexované atributy v kombinaci s negativně postavenými scénáři u obou zbývajících vyhledávacích skupin, dosáhneme tohoto kritické vyhledávání. Tento přístup může být předmětem dalšího vylepšení (možná vylepšení jsou uvedena na konci práce ve vlastní sekci). Níže zobrazený obrázek je znovu uveden v plném rozlišení jako příloha na konci této práce (konkrétně jako příloha Kompletní hierarchie zpracovávání).



Obrázek 22: Kompletní hierarchie vyhledávání

4.1.5 Zkrácené vyhodnocení a omezení velikosti výsledku

Kaskádové zpracování umožňuje použít zkráceného vyhodnocení. Jedná se o mechanismus umožňující zastavit vykonávání jednotlivých dotazů v okamžiku, kdy je jasné, že daný dotaz nebude mít dalšího užitku.

Užitkem v této situaci je myšlen stav, kdy počet vrácených řádků je větší než jedna. Představme si modelovou situaci, kdy zpracováváme druhou skupinu ze tří a výsledkem dotazu je prázdná množina výsledků. Pak je již zcela jasné, že i další krok také nic nevrátí. Je to z toho důvodu, že následující krok bude průnikem množiny výsledku třetí zpracovávané skupiny (ten, který teprve budeme vykonávat) a výsledku z množiny předchozího hledání (ten je ale prázdný).

Zde by již měla být vidět výše zmíněná výhoda oproti separátnímu zpracování. V separátním zpracování nedochází ke spolupráci či propojení předchozích výsledků. Proto je možné, že i když každá skupina vrátí nějaký nenulový výsledek, tak při finálním průniku bude výsledná množina prázdná (jednoduše neexistují společné vzorky pro jednotlivé skupiny). Skutečnost, že takové vzorky neexistují, poznáme až po vykonání všech skupin.

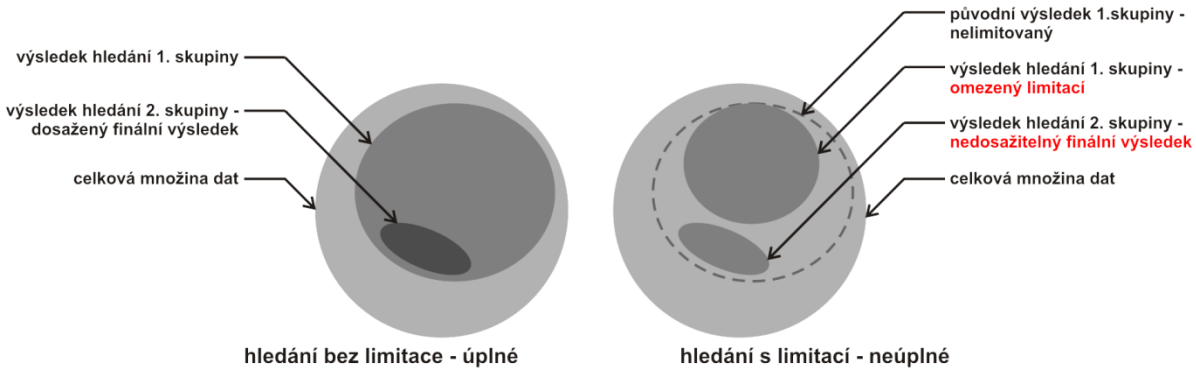
Další podstatnou částí diskutovanou v této sekci je omezení velikosti výsledku. Podobně jako kaskádové zpracování a z něho vycházející potenciální zkrácené vyhodnocení může i vhodné omezení značně ovlivnit délku provádění jednotlivých skupin.

V této fázi návrhu je potřeba velmi dobře uvážit, jestli nám jde o úplnost, nebo o rychlost. Oba tyto pohledy něco stojí. Úplnost stojí především čas, takže je v přímém rozporu s rychlostí. Z toho tedy vyplývá, že rychlost bude nějakým způsobem ovlivňovat právě úplnost.

V návrhu zpracování skupin je zakomponován jistý kompromis mezi úplností a rychlostí ve vztahu k omezení. Pokud je vyžadováno zpracování více než jedné skupiny, pak je třeba dostát naprosté úplnosti. To znamená, že dotazy z jiné než poslední skupiny nejsou omezeny velikostí vráceného výsledku (vrací se kompletní sada).

Tento, do jisté míry diskutabilní, postoj při vyhledávání je opodstatněn potřebou vrátit i takové vzorky, které jsou mimořádně specifické.

Celá podstata problému plyne ze skutečnosti, že nejsme schopni zaručit, že daná, nyní již limitovaná, skupina (prvních N vzorků splňujících požadavky první zpracovávané skupiny, kde N je celé kladné číslo) obsahuje dostatek vzorků s požadovanými vlastnostmi druhé skupiny. Dále bude ještě hůře zaručitelné, že v takto vzniklém průniku skupin (zpracované předchozí dvě skupiny) bude dostatek vzorků, které by měly vlastnosti skupiny tři. Pokud prvních N-1 skupin (kde N je počet vyhledávacích skupin) neomezíme, pak s jistotou dostaneme kompletní sadu možných výsledků a tudíž i ten velmi specifický vzorek (pokud vzorek existuje, tak jej poslední vykonávaná skupina najde vždy a může tak sestavit neprázdný průnik). Danou problematiku se pokouší ještě graficky znázornit následující obrázek.



Obrázek 23: Grafické znázornění úplnosti výsledku při použití limitace

Jelikož je výše v práci uvedeno, že se jedná o kompromis, je třeba doplnit i druhou stranu mince. Úplnost je potlačena v poslední skupině hierarchického vyhledávání. Tady je důležité si uvědomit, že velikost poslední skupiny bude podle návrhu přímo ovlivňovat reálnou velikost výstupu, který bude uživateli nabídnut. Výše v práci je tato myšlenka již popisována, ale zde bych ještě uvedl několik možností, jak je přece jen možné limit omezit.

Původní počet vzorků, který se zobrazuje na stránku, byl po diskuzi se zadavatelem stanoven na třicet záznamů. Z toho vyplývá, že je možné ukončit provádění poslední požadované skupiny již po třiceti nalezených vzorcích. Tento fakt se může značně projevit v úspoře času, zvláště pokud by se v případě absence omezení jednalo o velkou výslednou množinu.

Během návrhu metriky pro omezení se ovšem objevily hlasy, které požadovaly možnost zvýšení limitu pro některé speciální potřeby analýzy. I když s tím původní (výše popisovaný) návrh aplikace nepočítal, nakonec byly tyto požadavky do návrhu zakomponovány.

Změny spočívají v přidání možnosti na zvýšení limitu, který ovšem obsahuje strop. Je to z toho důvodu, že je zde stále zcela nemyslitelné, aby byl počet vzorků, které je možno vrátit, zcela neomezený. Tento nový maximální limit byl stanoven na 999 vzorků. Použití rozšířeného limitu je explicitní, takže uživatel, který nepotřebuje více vzorků, nemusí nic měnit.

Negativní rysy úplnosti

Výše navrhovaný postup úplnosti může ovšem značně ovlivnit efektivitu vyhledávání v případě, že první stupeň kaskády povede na velkou výslednou množinu dat. Tento „rys“ je třeba mít na paměti a na základě výsledků způsobů užití (vznikne na základě logu) možná bude nezbytné implementovat slovník (v podobě tabulky v databázi), který nám umožní tyto „velkoobjemové“ atributy identifikovat a při jejich výskytu přistoupit k běžnému sekvenčnímu zpracování bez použití kaskády. Tím pádem bychom pak mohli případný negativní vliv omezit. Vhodným způsobem, jak tento „slovník“ potenciálně implementovat, je využití informací tabulky záznamů, která bude vznikat po každém úspěšném hledání a nebo pro tyto informace vytvořit dedikovanou tabulku, která by funkční jedinečností urychlila hledání inkriminovaných atributů.

4.1.6 Struktura navrhovaných dotazů SQL

Na následujících řádcích této sekce budou popsány jednotlivé konstrukce SQL, které byly navrženy a později i implementovány v systému vyhledávání vzorků. Vzhledem k různorodosti jednotlivých dotazů SQL budou tyto konstrukce popsány v rámci jednotlivých scénářů daných skupin. V sekci detekcí bude pak blíže specifikována jistá část optimalizace, která umožňuje minimalizovat počet dotazů při určité podobnosti v rámci jednotlivých detekčních skupin.

Skupina strukturálních indexů – pozitivní scénář

Tento scénář je navržený jako sada vnitřních spojení (`INNER JOIN`) požadovaných a v tabulce `ctrl_avgmis` definovaných tabulek SQL. Jelikož, dle návrhu, dochází ke spojení vazebních tabulek přes sloupec `id_file` (identifikátory do bázové tabulky `files`), je možné získat z tohoto spojení právě hodnotu `id_file`, aniž bychom požadovali další data. Tím se omezí přístup k nepotřebným sloupcům, přesně jak je popisováno ve výčtu možných optimalizací.

Pokud jsme již schopni získat data z tabulek (konstrukcí SQL `INNER JOIN`) můžeme dále přistupovat k optimalizaci podmínkové části. V tomto ohledu jsou zajímavé pouze potenciální vícenásobné hodnoty („archivátory“, „packery“ a skupiny). Z vícenásobných hodnot je třeba rozlišovat mezi vztahy, které tyto hodnoty vůči sobě mají. Jelikož skupiny mají mezi sebou logický vztah `AND`, pak zde návrh počítá s konstrukcí SQL `AND` mezi jednotlivými hodnotami. Alespoň minimální optimalizace skupin spočívá v nepřipojování relativně malého číselníku (tabulka `c_groups`) v sekci spojených tabulek, ale počítá s přímým překladem těchto hodnot už na úrovni grafického uživatelského rozhraní (angl. GUI). Důkaz, že tento postup je efektivnější, ačkoliv je připojovaný slovník malý, je uveden později v testech.

Při vazbě OR, kterou mezi sebou mají vícenásobné výskyty „archivátorů“ a „packerů“, byla zvolena konstrukce SQL `IN`. I zde se tedy snažíme využít rady pro optimalizaci dotazů. Nedá se předpokládat, že by počet „archivátorů“ či „packerů“ byl tak velký (vzhledem k současné velikosti jejich slovníků), že se tento postup stane naopak velmi neefektivním (viz problematika popsaná v optimalizacích dotazů výše v textu).

Ještě před uvedením ukázky bych rád zmínil, že ve skupině strukturálních indexů se nachází jeden atribut umístěný v tabulce files. Jedná se o secthash. Secthash sama o sobě je velmi specifickým ukazatelem na některé vzorky. Jestli má být omezování velikosti kaskády efektivní, pak se musíme snažit tento atribut protlačit do první zpracovávané skupiny.

Pro lepší představu je níže v textu uveden ukázkový dotaz SQL pro pozitivní scénář strukturálního indexu. V tomto dotazu je možné vidět jak výskyt jedné skupiny (group), tak i vícenásobný požadavek na „archivátory“ a „packery“ (řádky vyznačené červenou barvou). Hodnoty jsou sice reálné, ale takový vzorek v databázi neexistuje. Jedná se pouze o ilustraci argumentů.

```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=MyISAM
      SELECT r1.id_file FROM r_files_certificates r1
            INNER JOIN c_certificates c1 ON r1.id_certificate = c1.id_certificate
            INNER JOIN r_files_groups r2  ON r1.id_file = r2.id_file
            INNER JOIN r_file_archiver r3 ON r1.id_file = r3.id_file
            INNER JOIN c_archivers c3 ON r3.id_archiver = c3.id_archiver
            INNER JOIN r_files_iconhash r4  ON r1.id_file = r4.id_file
            INNER JOIN c_iconhash c4 ON r4.id_iconhash = c4.id_iconhash
            INNER JOIN r_file_packer r5 ON r1.id_file = r5.id_file
            INNER JOIN c_packers c5 ON r5.id_packer = c5.id_packer
      WHERE c1.SHA256='f8a79891fa481c2d96890985d2dc09d0a29025e7fd8435d6e88eb61efbe4c0b2'
            AND r2.id_group=3
            AND c3.archiver IN ('Generic CAB SFX','Rsrc-Package' )
            AND c4.iconhash='0000292aefe3527146d61e6167829e5e'
            AND c5.packer IN ('PE_Patch.UPX', 'UPX');
```

Obrázek 24: Ukázka dotazu SQL pro skupinu strukturálních indexů – pozitivní scénář

Skupina strukturálních indexů – negativní scénář

Negativní scénář je vzhledem k možnostem konstrukcí SQL možno řešit různými způsoby. Pro tento konkrétní skupinový scénář byla zvolena konstrukce SQL `DELETE FROM`. Jedná se o prosté mazání řádků z tabulky, které splňují požadovanou podmínku. Existuje několik důvodů, proč je tato varianta právě tou efektivní možností.

Prvním z těchto důvodů je skutečnost, že ačkoli požadujeme negativní scénář, tak jej ve skutečnosti negativně vůbec nevykonáváme. Toto trochu protichůdné tvrzení je vysvětleno podstatou vytvářené konstrukce SQL. Návrh se snaží ubránit použití konstrukcí SQL, jakou je `NOT EXISTS`. Místo toho je jeho podstatou převedení podmínkové části do pozitivního scénáře. Celá myšlenka spočívá v postupu, že nehledáme vzorky, které nemají danou vlastnost, ale naopak hledáme vzorky, které tuto vlastnost mají. Takové vzorky jsou pak vymazány z dané tabulky, čímž je dosaženo požadovaného výsledku.

Další pozitivem tohoto přístupu je programová kompatibilita s pozitivním scénářem (při tvorbě dotazu je možné použít jen trochu upravený algoritmus). Pokud se pozorně podíváme na ukázkový dotaz SQL řešící požadavek na záporný scénář, pak uvidíme, že je daná podmínková část naprosto stejná jako u pozitivního scénáře (pro každý jeden atribut). Pokud se podíváme na rozdíly, pak jedním z rozdílů je začátek dotazu. Tedy klauzule `DELETE FROM`. Posledním rozdílem od pozitivního scénáře je skutečnost, že musíme tento dotaz provést pro každý atribut. Je to z toho důvodu, že nechceme vzorky, které mají jakoukoli z požadovaných vlastností negativního hledání (nejen vzorky, které obsahují všechny tyto vlastnosti).

Posledním zmíněným pozitivem je fakt, že nemusíme vytvářet novou dočasnou tabulku (výsledek dalšího stupně kaskády) a plnit ji nalezenými daty. Podle návrhu bude stačit jen pozměnit název aktuálního úložiště pro další stupeň zpracování (či pro zveřejnění výsledku, pokud tento stupeň byl poslední).

Tento odstavec ještě vysvětluje, proč není v takovémto dotazu použita tzv. „salámová metoda“, která je popisována výše v práci. Tento přístup ji totiž fakticky nepotřebuje. Nezapomeňme, že data jsou ukládána do dočasných tabulek. V jazyce MySQL mohou s dočasnými tabulkami

pracovat jen instance připojení, které tuto tabulku vytvořily. A jelikož má salámová metoda omezit blokování přístupu pro další připojení, pak tuto skutečnost nemusíme vůbec zohledňovat.

Poslední částí návrhu zpracování negativního scénáře je úsek, který popisuje nakládání s atributy globálních zákazů „archivace“ a „packování“ (požadavky na vzorky, které nejsou těmito softwary vůbec postiženy). Způsoby jak tento problém řešit jsou v zásadě dva.

Prvním z nich je spojení SQL `LEFT JOIN` a podmínka dále pokračuje požadavkem na vzorky, které nemají hodnotu `NULL` (tedy v kontextu scénáře smaž všechny, které mají nějaký záznam).

Druhým způsobem, a pro budoucí program lepším, je ovšem konstrukce SQL `INNER JOIN`, která již dále nepotřebuje podmínkovou část (v úvahu se vezmou jen ty vzorky, u kterých je možné spojení – takže se tabulka nemusí procházet vůbec). Tento způsob je i rychlejší, byť zanedbatelně. Proto se stal součástí návrhu na vyhledávání s požadavkem globálních zákazů na „archivaci“ a „packování“.

```
DELETE tmp
      FROM struct_tmp_table tmp
            INNER JOIN r_file_archiver r0 ON tmp.id_file = r0.id_file
            INNER JOIN c_archivers c0 ON r0.id_archiver = c0.id_archiver
      WHERE
            c0.archiver = 'Generic CAB SFX';

DELETE tmp
      FROM struct_tmp_table tmp
            INNER JOIN r_files_groups r0 ON tmp.id_file = r0.id_file
      WHERE
            r0.id_group=1;
```

Obrázek 25: Ukázka dotazu SQL pro skupinu strukturálních indexů – negativní scénář

Skupina strukturálních indexů – smíšený scénář

Smíšený scénář u skupiny strukturálních indexů využívá oba svoje sourozence. Daný požadavek se musí rozložit na pozitivní a negativní část. Prvně je provedena pozitivní část (pozitivní scénář) a poté následuje negativní část (negativní scénář), která jen upraví výslednou podobu dočasné tabulky vzniklé v pozitivní části. Jelikož se jedná pouze o zřetězení těchto dvou scénářů, tak zde není uveden žádný dotaz SQL.

Skupina files – pozitivní scénář

Atributy skupiny `files` dle požadavků zadavatele neumožňují jiný než pozitivní scénář. Jelikož jsou v dané skupině jen atributy pocházející z tabulky `files`, tak celá konstrukce dotazu SQL se bude zabývat vyhledáváním nad jedinou tabulkou.

Po stránce návrhu konstrukce dotazu SQL byla tedy zvolena varianta, kdy na počátek podmínkové části jsou řazeny ty sloupce, které jsou indexovány. To by mělo podle [1] umožnit vyhledávání po indexu. Jednotlivé atributy mezi sebou mají logickou vazbu `AND`. Ukázka typického dotazu SQL skupiny `files` je níže.

```
CREATE TEMPORARY TABLE files_tmp_table ENGINE=MyISAM
      SELECT f.id_file
            FROM files f
      WHERE
            f.STATUS=2 AND
            f.VERINFO_FileDescription = 'xxx%' AND
            f.VERINFO_CompanyName = 'xxx%' AND
            f.VERINFO_LegalCopyright = 'xxx%' AND
            f.VERINFO_FileVersion = 'xxx%' AND
            f.SUBTYPE = 0 AND
            f.FILESIZE = 10 AND
            f.FIRST_SEEN = '2011-04-02 10:00:00' AND
            f.TYPE = 1;
```

Obrázek 26: Ukázka dotazu SQL pro skupinu files – pozitivní scénář

Skupina detekcí – pozitivní scénář

Pro zpracovávání pozitivního scénáře skupiny detekcí (pro jednu detekční skupinu v aplikaci `AVGMIS` nebo její pozitivní část) byl v rámci návrhu zvolen stejný princip jako u strukturálních indexů. Opět se jedná o soustavu vnitřních spojení tabulek SQL (`INNER JOIN`). V tomto ohledu se v dané skupině nejedná o žádnou změnu. Vnitřní spojení se během testů projevilo jako efektivní.

Specifika jsou obsažena až v možnostech, s jakými lze vyhledávat detekce. Pokud hledáme přímou shodu názvu, pak lze použít v podmínkové části pouhé „rovná se“ k zadané hodnotě. Metoda zvaná otevřený konec využívá klauzuli `LIKE` a pokračuje hodnotou doplněnou procentem. Podřetězec je podobným případem, jen jsou procenta umístěna na začátku i na konci. Uměle vytvořený (nereálný), ale syntakticky kompletní, dotaz SQL, který využívá všech tří metod je k vidění níže v textu. Tento dotaz představuje zpracování jedné detekční skupiny.

```
CREATE TEMPORARY TABLE detection_tmp_table ENGINE=MyISAM
      SELECT SQL_NO_CACHE r.id_file
            FROM r_files_det_avg r
            INNER JOIN c_det_avg c ON r.id_det_avg = c.id_det_avg
      WHERE
            c.det_avg = 'Adware AdSearcher.B' AND     /* exact match */
            c.det_avg LIKE 'Adware AdSearcher%' AND   /* open end */
            c.det_avg LIKE '%AdSearcher%';            /* substring */
```

Obrázek 27: Ukázka dotazu SQL pro skupinu detekcí – pozitivní scénář

Skupina detekcí – negativní scénář

Jelikož na rozdíl od skupiny strukturálních indexů není požadováno, aby skupina detekcí byla schopna hledat vzorky dle jména, tak není třeba řešit situaci podobnou jako v případě strukturálních indexů.

Připomeňme si, že zadavatel požaduje pouze vyhledávání vzorků, které nejsou detekovány daným výrobcem. V tom případě stačí vhodně použít právě konstrukce levého spojení (`LEFT JOIN`) tabulky hledaného výrobce a odkazovat se na vzorky, které mají hodnotu identifikátoru vzorku (`id_file`) ve vazební tabulce `NULL`.

Současně tento přístup nese s sebou další pozitivum. Není třeba připojovat celý detekční slovník (není třeba hledat název), čímž získáme další úspory. Ukázka dotazu SQL je uvedena níže.

```
CREATE TEMPORARY TABLE detection_tmp_table ENGINE=MyISAM
      SELECT SQL_NO_CACHE f.id_file
            FROM files f
            LEFT JOIN r_files_det_avg r0 ON f.id_file = r0.id_file
      WHERE
            r0.id_file IS NULL;
```

Obrázek 28: Ukázka dotazu SQL pro skupinu detekcí – negativní scénář

Skupina detekcí – smíšený scénář

Smíšený scénář je jako v předchozích případech kombinací pozitivního a negativního scénáře. Tentokrát ale není třeba rozkládat dotaz SQL do dvou oddělených dotazů (dotaz s klauzulí `INNER JOIN` a pak dotaz s příkazem `DELETE FROM`). Díky společné povaze obou spojení (syntaktické podobě) je možné tento požadavek vykonat současně v rámci jednoho dotazu.

V některých případech, které jsou popsány v následující sekci, je rozumnější provést požadavky odděleně. Podobně jako je tomu ve skupině strukturálních indexů.

```
CREATE TEMPORARY TABLE detection_tmp_table ENGINE=MyISAM
      SELECT SQL_NO_CACHE r0.id_file
            FROM r_files_det_avg r0
            INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
            INNER JOIN r_files_det_kav r1 ON r0.id_file = r1.id_file
            INNER JOIN c_det_kav c1 ON r1.id_det_kav = c1.id_det_kav
            LEFT JOIN r_files_det_mcafee r2 ON r0.id_file = r2.id_file
      WHERE
            c0.det_avg = 'Virus found BackDoor.ASP' AND
            c1.det_kav = 'Backdoor.ASP.Ace.ah' AND
            r2.id_file IS NULL;
```

Obrázek 29: Ukázka dotazu SQL pro skupinu detekcí – smíšený scénář – bez společné pozitivní částí

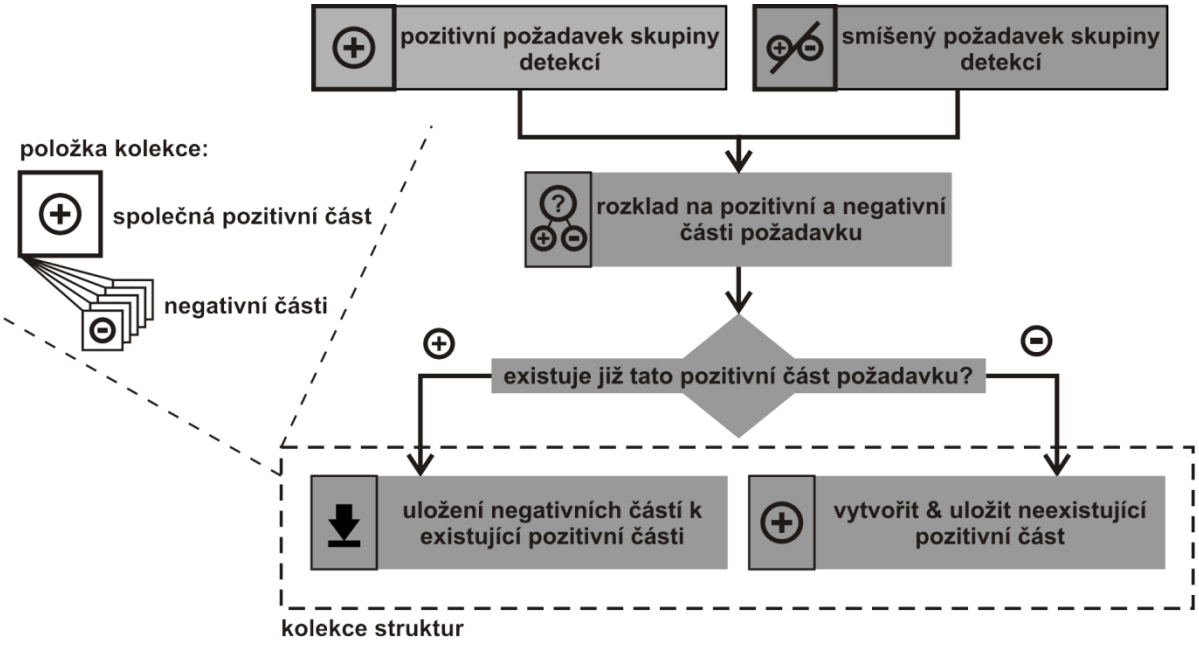
```
CREATE TEMPORARY TABLE temp_detection_tmp_table ENGINE=MyISAM     /* common pos. part */
      SELECT SQL_NO_CACHE r0.id_file
            FROM r_files_det_avg r0
            INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
            INNER JOIN r_files_det_kav r1 ON r0.id_file = r1.id_file
            INNER JOIN c_det_kav c1 ON r1.id_det_kav = c1.id_det_kav
      WHERE
            c0.det_avg LIKE 'Virus found BackDoor%' AND
            c1.det_kav LIKE 'Backdoor.ASP.Ace%';
-------------------------------------------------------------------------------
INSERT IGNORE INTO detection_tmp_table    /* mixed group #1, neg. part */
      SELECT SQL_NO_CACHE tmp.id_file
            FROM temp_detection_tmp_table tmp
            LEFT JOIN r_files_det_kav r0 ON tmp.id_file = r0.id_file
      WHERE
            r0.id_file IS NULL;
-------------------------------------------------------------------------------
INSERT IGNORE INTO detection_tmp_table    /* mixed group #2, neg. part */
      SELECT SQL_NO_CACHE tmp.id_file
            FROM temp_detection_tmp_table tmp
            LEFT JOIN r_files_det_mcafee r0 ON tmp.id_file = r0.id_file
      WHERE
            r0.id_file IS NULL;
```

Obrázek 30: Ukázka dotazu SQL pro skupinu detekcí – smíšený scénář – se společnou pozitivní částí

Ukázka dotazu SQL pro tento scénář je uvedena výše. V tomto případě se jedná o první (sloučený) tvar dotazu. Současně si povšimněte, že se zde (u všech zpracování skupiny detekcí) využívá klauzule `INSERT INTO` místo doposud používané klauzule `CREATE TEMPORARY TABLE`. Je to z toho důvodu, že každá skupina může přinést jinou množinu vzorků, které nutně nemusejí být v průniku s jinými detekčními skupinami (skupiny mají mezi sebou vztah `OR`).

Skupiny detekcí – optimalizace

Předposlední sekce návrhu zpracování dotazů se zabývá optimalizací požadavků na atributy ve skupině detekcí. Jelikož jsou mezi sebou detekční skupiny v logické vazbě `OR`, pak se jejich výsledky mohou do jisté míry potenciálně překrývat. Tato optimalizace má za úkol najít příležitostné překryvy a odstranit je.

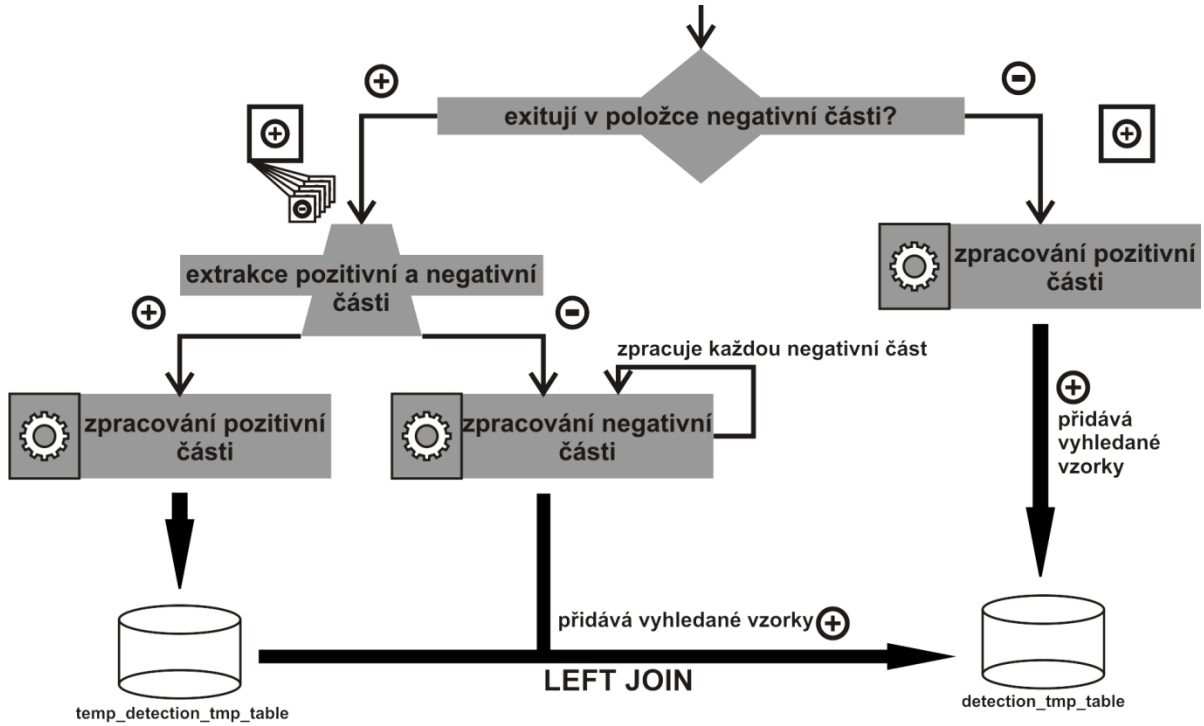


Obrázek 31: Diagram znázorňující hledání společných částí

Původní úmysl provádět nejen optimalizaci dotazů ve formě efektivnější restrukturalizace, ale i používat tento detektor překryvů, je opodstatněn na základě požadavků zadavatele. Počet detekčních skupin není v podstatě omezen a s rostoucím počtem těchto skupin může docházet ke zvýšení potenciálních překryvů nad těmito detekčními skupinami.

Představme si situaci, kdy uživatel požaduje vzorky A výrobce B a pak požaduje v rámci jedné detekční skupiny i vzorky, které nejsou detekovány výrobcem C. Pokud v další detekční skupině požaduje opět vzorky A výrobce B, teď už bez druhé části, pak je jasné, že tyto výsledné skupiny se budou překrývat. Druhá detekční skupina vrátí jistou množinu vzorků, v níž jsou vzorky první detekční skupiny podmnožinou. V takovém případě stačí provést jen druhou (silnější) detekční skupinu a tím ušetřit celý smíšený dotaz.

Uvedené diagramy (níže a výše) znázorňují návrhy možných optimalizací detekčních skupin v systému AVGMIS. V prvním diagramu je vyobrazeno hledání společných částí – tedy překryvů. V druhém diagramu je znázorněno, jakým způsoben je navrženo vykonávání takto předzpracovaných dotazů. Současně si všimněte, že jsou využívány obě dočasné tabulky pro skupiny detekcí.



Obrázek 32: Diagram znázorňující vykonávání optimalizovaných skupin detekcí

4.1.7 Typy dočasných úložišť

Tato poslední sekce si klade za úkol popsat typy dočasných úložišť. V původním návrhu systému byla volba typu dočasných tabulek jasná. Jednalo se o typ MEMORY. Tento tabulkový typ se jevil jako nejvhodnější varianta z důvodu místa, kam je daná tabulka ukládána. Podle [1] a také podle [3] se tabulky typu MEMORY ukládají přímo do paměti, a ne na disk. Tím bychom získali další úspory času během vykonávání dotazů.

Ovšem podstatným negativem je limit velikosti těchto tabulek. Bohužel, i když návrh systému počítá s ukládáním pouze jednoduchých celočíselných identifikátorů vzorků, jejich počet není nijak omezen. To se během testování projevilo chybou při vykonávání, neboť tohoto limitu bylo dosaženo.

Jako odpověď na tuto vlastnost byly dočasné tabulky změněny na typ MyISAM. MyISAM sice nepodporuje transakce, ale pro dočasné tabulky a jejich využití v systému AVGMIS to nijak nevadí. Pozitivní na celém typu je skutečnost, že vždy přesně zná celkový počet řádků v tabulce. Tento fakt je později využit při protokolování a následné analýze velikosti jednotlivých stupňů kaskády, aniž by vznikla újma na rychlosti zpracování.

Tímto posledním odstavcem končí návrh zpracování a tvorby dotazů v rámci systému AVGMIS. Všechny testy dotazů SQL jsou přiloženy na disku DVD u diplomové práce. Zde je také možné prohlédnout si původní znění algoritmů a jejich diagramů tak, jak byly vytvářeny.

4.2 Grafické uživatelské rozhraní

Součástí systému na vyhledávání vzorků bude i grafické uživatelské rozhraní, které by mělo umožňovat pohodlné zadávání hledaných atributů. Po provedeném hledání musí uživatelské rozhraní takto nalezené vzorky přehledně nabídnout uživateli. Tato podkapitola se tedy zabývá návrhem takového grafického uživatelského rozhraní. Současně nabídne i pohled na předlohu, na jejímž základě bude rozhraní navrhováno.

4.2.1 Aplikace VTMIS – předloha aplikace AVGMIS

Před vlastním návrhem uživatelského rozhraní AVGMIS je uveden popis uživatelského rozhraní systému VTMIS. Tento systém umožňuje podobně jako systém AVGMIS vyhledávat vzorky v databázi. VTMIS, neboli Virus Total Malware Intelligence Service, je aplikace vyvíjená společností Virus Total, která současně provozuje „mass virus center“ pro testování vzorků. Tato centra jsou vlastně servery, které sdružují několik antivirových strojů (momentálně v řádu desítek). Tyto stroje pak nabízejí uživatelům služby ve formě skenování vložených vzorků – tedy uživatel dostane možnost prověřit vložený vzorek všemi antivirovými stroji.

Rozhraní příkazové řádky

Tato forma vstupu zpracovává požadavky na vyhledávání ve formě jednoduchých tokenů. Tokenem je zde myšlena dvojice atribut a hodnota. Výhodou takového vstupu je nepotřeba mít rozsáhlý formulář, kde by možné všechny požadované atributy vyplnit (uživatel zadává jen tokeny).

Ačkoli zadavatel přímo nepožadoval tento druh vstupu, přesto bude v systému AVGMIS navržen a později i implementován. Současná podoba rozhraní příkazové řádky systému VTMIS je uvedena na obrázku níže. Navrhovaná podoba příkazové řádky pro systém AVGMIS je pak popsána dále v textu.

Obrázek 33: Grafická podoba příkazové řádky systému VTMIS

Plné grafické rozhraní

Plné grafické rozhraní je onen výše zmíněný formulář, který obsahuje kompletní funkčnost příkazové řádky. Tento formulář je nabízen ve formě nejrůznějších textových vstupů, zaškrtávacích políček

(angl. checkbox), rozevíracích nabídek (angl. select či drop down menu) a jiných užívaných prvků HTML (VTMIS je webová aplikace).

Výhodou takového rozhraní je nepotřeba znalosti jazyka tokenů, který řídí vyhledávání pomocí příkazové řádky. Dále pak je možné toto rozhraní více přizpůsobit potřebám relativně neznalého uživatele (vytvořit jej více intuitivní) a umožnit mu tak pohodlnější vyhledávání vzorků. Plné grafické rozhraní aplikace VTMIS je vloženo v příloze v této práci (konkrétně jako Aplikace VTMIS – rozšířený vstup).

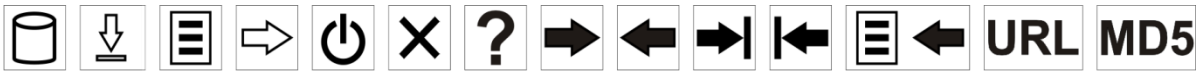
4.2.2 Styl navrhovaného grafického uživatelského rozhraní

Styl grafického rozhraní navržený pro tento systém podléhal několika požadovaných specifikům. Hlavním požadavkem byla jednoduchost s minimálním počtem efektů. Tento požadavek vyplýval ze skutečnosti, že k danému systému mnohdy budou uživatelé přistupovat přes vzdálenou plochu svých pracovních stanic.

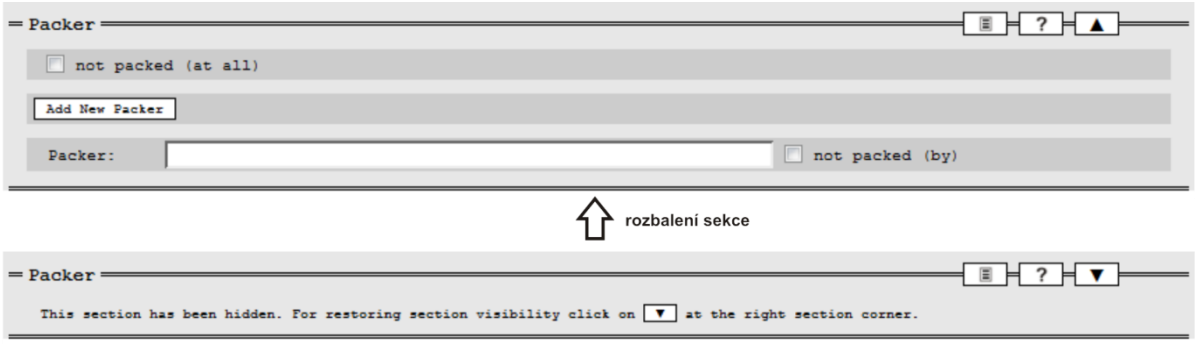
Dalším požadavkem byla snadná grafická rozšiřitelnost, která by měla umožnit vložit nový atribut hledání jednoduše a efektivně bez potřeby výrazného zásahu do grafického rozhraní. Není vyloučené, že systém může do budoucna růst, takže i v tomto ohledu je potřeba na tyto požadavky brát ohled (ve skutečnosti bylo potřeba po dokončení implementace doplnit ještě atribut pro limitaci výsledku, takže tento požadavek byl testován v praxi).

Posledním zde zmiňovaným požadavkem je co nejvyšší možná přehlednost. Uvažme, kolik atributů je možné vyhledávat a jakých četností mohou nabývat. V tomto ohledu je tedy potřeba přemýšlet o skladbě jednotlivých sekcí v rozšířeném uživatelském rozhraní.

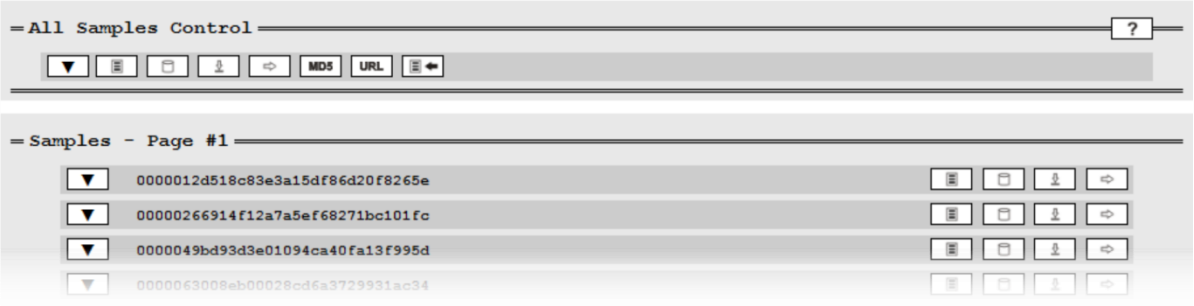
Následující obrázky ukazují některé navrhované podoby designu aplikace AVGMIS. Schválně zde chybějí obrázky ukazující celá řešení (zvýšená četnost atributů, pohledy na celá uživatelská rozhraní) aplikace AVGMIS. Tato řešení jsou popisována a zobrazována dále v práci.



Obrázek 34: Ukázka návrhu funkčních ikon



Obrázek 35: Ukázka návrhu rozbalené a zabalené sekce – řeší úsporu místa při nevyužívaných atributech



Obrázek 36: Ukázka návrhu výstupu jednotlivých vzorků

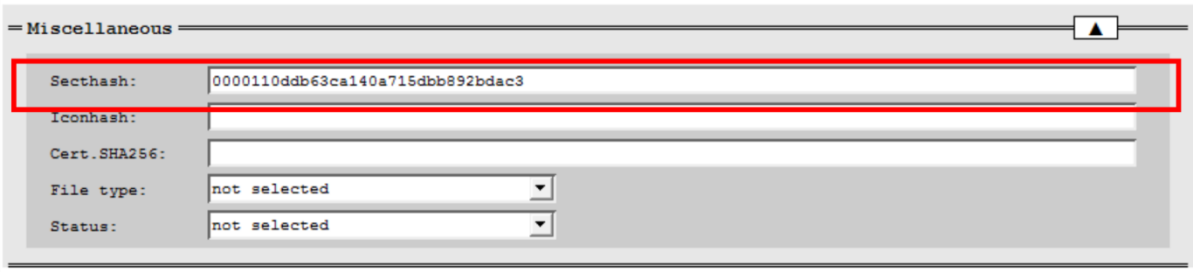
4.2.3 Vstup

Návrh počítá se třemi vstupními rozhraními. Jedná se o rozhraní příkazové řádky (dále jako `cmd`), rozhraní rozšířeného vstupu (dále jako `advanced`) a konečně rozhraní přímého vstupu (vzniklo jako reakce na opakované hledání bez nutnosti použít grafické rozhraní).

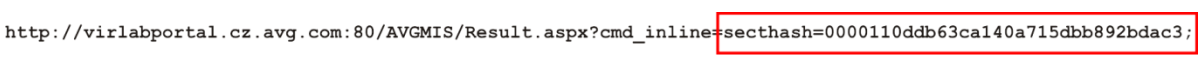
Vstupem příkazové řádky je syntakticky správný řetězec tokenů. Vstupem rozšířeného vstupu jsou vyplněné elementy jazyka HTML a ASP.NET. A vstupem přímého vstupu je adresa URL se zakomponovaným řetězcem „`command string`“. „`Command string`“ je označení pro výše zmíněný řetězec tokenů, který je zcela shodný s formou (syntaxí a sémantikou) vstupu rozhraní příkazové řádky.

Následující obrázky ukazují návrhy grafického rozhraní pro jeden specifický vyhledávaný atribut (`secthash`). Všechny tři níže uvedené obrázky jsou ekvivalentními vstupy.

Obrázek 37: Grafické znázornění vstupu cmd



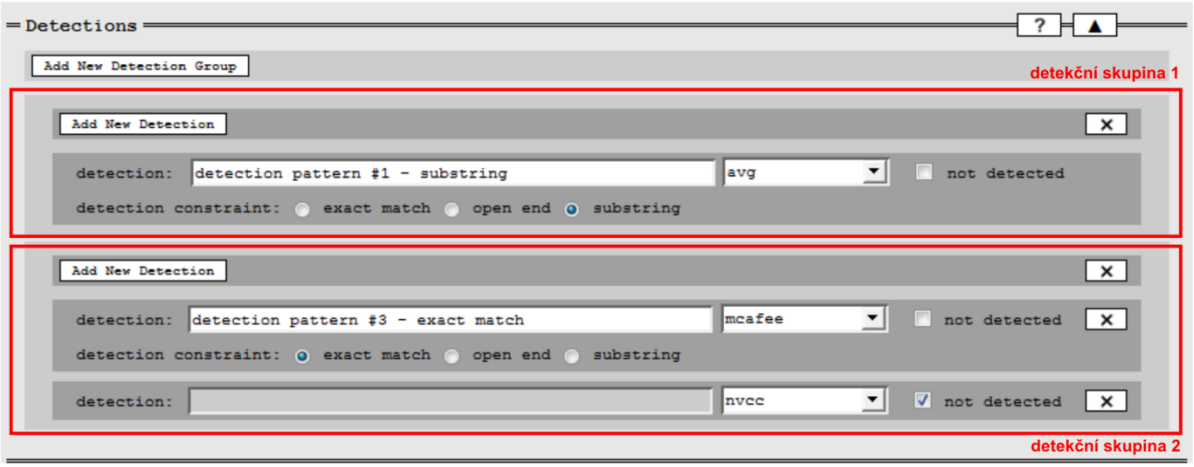
Obrázek 38: Grafické znázornění vstupu advanced



Obrázek 39: Grafické znázornění přímého vstupu

Požadované atributy a jejich možné četnosti

Vzhledem k relativnímu opakování problematiky u jednotlivých atributů zde budou uvedeny jen grafické návrhy sekce detekcí. Tato sekce obsahuje obě diskutované problematiky (potenciální vysokou četnost a „záporný“ vstup).



Obrázek 40: Grafický návrh detekčních skupin v rozšířeném vstupu

Na výše uvedeném obrázku je vidět dvě detekční skupiny. První z nich obsahuje jeden „kladný“ požadavek na detekci a druhá detekční skupina má dva požadavky na detekci (z toho jeden je záporný).

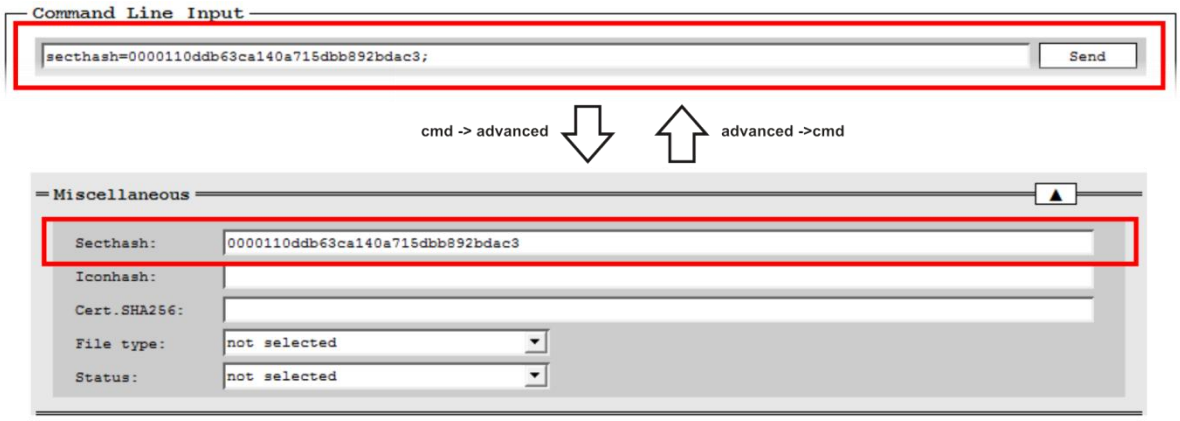
Díky tomuto systému zobrazení je pak možné dosáhnout libovolného počtu detekcí uvnitř detekčních skupin. Stejně tak i libovolného počtu vlastních detekčních skupin.

Na základě tohoto návrhu je pak možné nasimulovat široké spektrum možností, které je ostatně požadováno zadavatelem v této oblasti. Ostatní požadované atributy využívají stejného návrhu jako detekce.

Přepínání mezi jednotlivými rozhraními

Jelikož návrh počítá se dvěma grafickými druhy vstupů, je nezbytné umožnit uživateli funkci pro jejich přepínání. Celá podstata spočívá v tom, že rozšířený vstup při přepnutí do vstupu příkazové řádky musí data převést a přeformátovat (hodnoty v elementech jazyka HTML jsou přetransformovány do odpovídajícího řetězce tokenů).

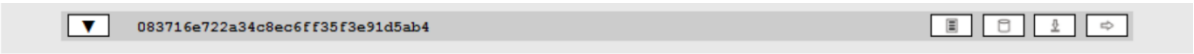
Překlad a přepínání je podle návrhu funkcionalitou oboustrannou, což v konečném důsledku umožňuje vytvořit požadavek v rozšířené nabídce a transformovat jej do „skladnější“ podoby příkazové řádky (tak jak je ukázáno na obrázku níže).



Obrázek 41: Grafické znázornění překladu rozšířeného vstupu na vstup příkazové řádky

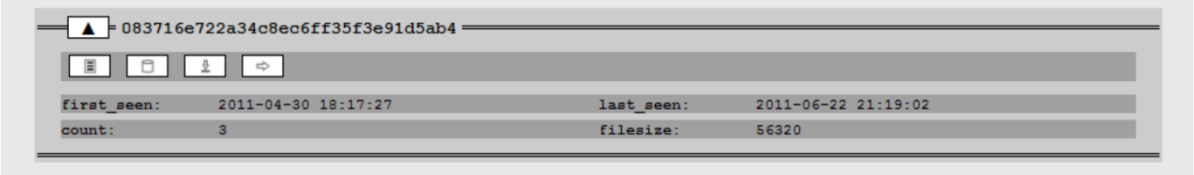
4.2.4 Výstup

Návrh výstupu, kde jsou zobrazovány jednotlivé vzorky, opět vyžaduje především jednoduchost a přehlednost. Daný vzorek, poté co je vrácen, se dá rozbalit to do tří podob. První podobou je pouze jeho hash md5 a v seznamu výstupů vytváří jeden řádek dané výsledné množiny.



Obrázek 42: Grafické znázornění první úrovně výstupu

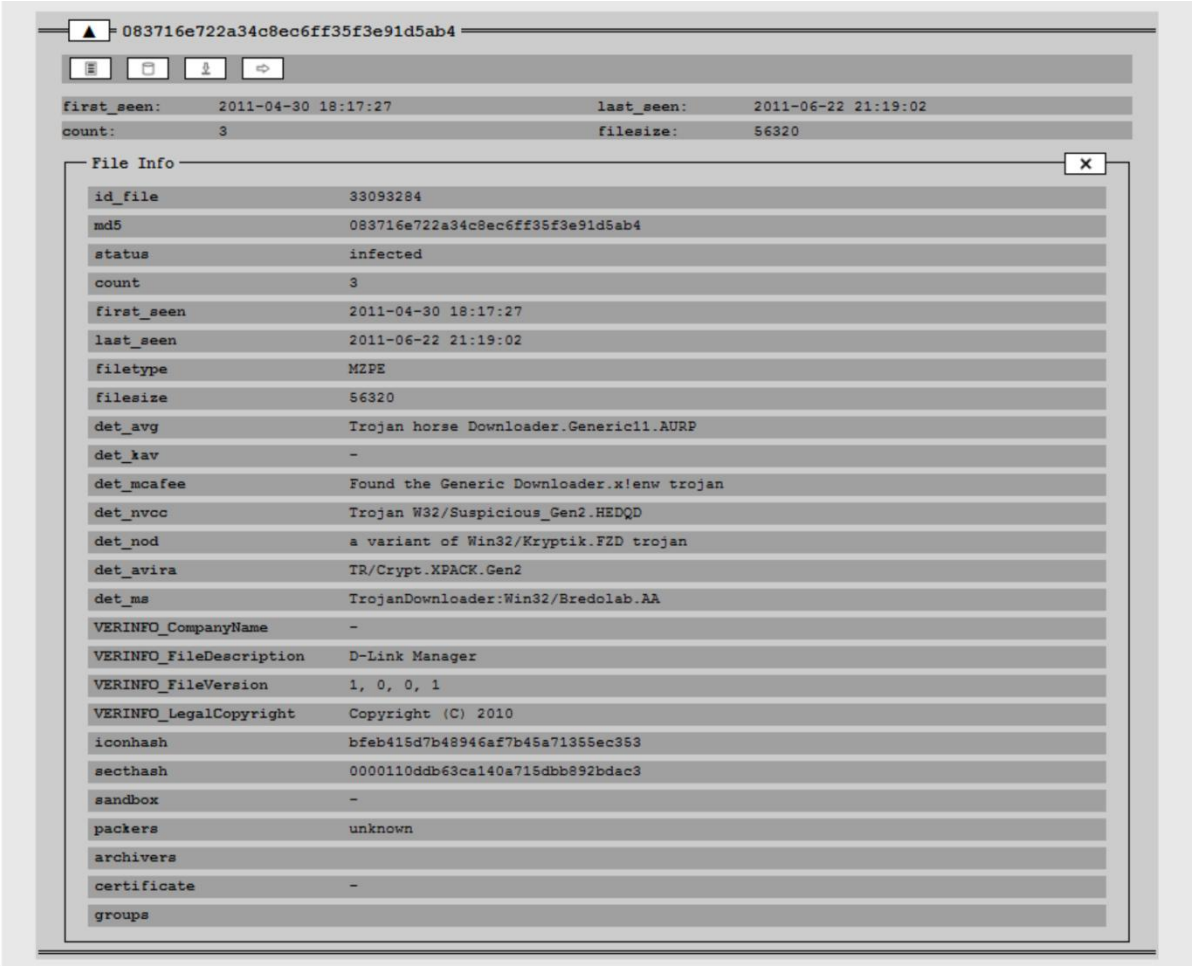
Druhou úrovní je podle návrhu už trochu rozšířenější a na informace bohatější podoba. Jsou zde zachyceny základní informace o velikosti a čase, kdy jsme se s daným vzorkem setkali poprvé. Tato úroveň slouží jako místo, kde je možné získat první zajímavé (ale ne nejpodrobnější) informace o daném vzorku.



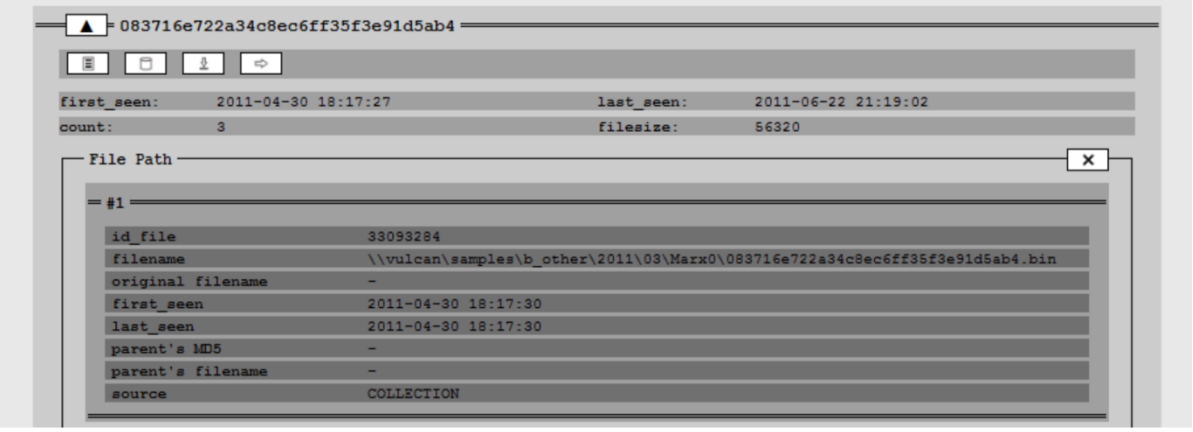
Obrázek 43: Grafické znázornění výstupu druhé úrovně

Třetí, a nejpodrobnější úroveň umožňuje zobrazit většinu dostupných informací, které jsou o vzorku k dispozici. Tato úroveň se dělí na dvě části. První částí jsou detailní informace o vzorku (kdo jej detekuje a jak, jestli je zabalen či archivován, jestli patří do nějaké skupiny a mnohé další).

Druhou částí jsou informace z úložiště, jako je zdroj, kde je vzorek uložen apod. Pokud vzorek není uložen, nejsou zde zobrazeny žádné informace.



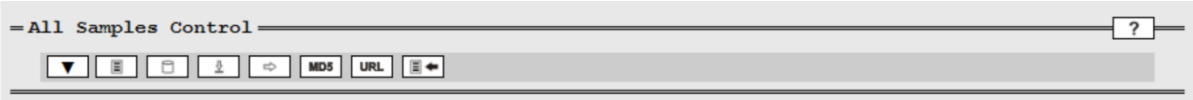
Obrázek 44: Grafické znázornění výstupu třetí úrovně – detailní informace



Obrázek 45: Grafické znázornění výstupu třetí úrovně – informace z úložiště

Návrh výstupu je utvořen tak, aby se dalo přistoupit ke kterékoli informační úrovni. Dále je pak počítáno s možností zpětné editace požadavků, získání seznamu hashí md5 vyhledaných vzorků (možné použití v dalších interních nástrojích) či přístupu k danému výsledku bez uživatelského rozhraní prostřednictvím adresy URL (popisováno výše).

Pokud je třeba pracovat se všemi nalezenými vzorky současně, pak je k dispozici tzv. „globální ovládání“, které umožňuje funkcionalitu jednotlivých vzorků a přidává i tu výše zmíněnou (seznam hashí md5, nápověda, …).



Obrázek 46: Grafické znázornění návrhu globálního ovládání

5 Implementace systému AVGMIS

Tato kapitola se zabývá implementací výše uvedeného návrhu systému na vyhledávání vzorků. V následujících podkapitolách bude popsána programová struktura celého systému a dále pak v rámci další podkapitoly rozbor zpracování datových struktur, počínaje uživatelským požadavkem a konče proveditelným dotazem SQL.

Než budou popsány implementační detaily, je třeba alespoň ve stručnosti popsat vývojové prostředí a použité platformy. Celý systém vyhledávání je postaven na platformě ASP.NET od společnosti Microsoft. Jako „code-behind“ (programovací jazyk, který provádí webový server) byl zvolen jazyk C# důkladně popsaný na [8]. Bylo to z toho důvodu, že je syntakticky do jisté míry podobný jazyku C/C++ a obsahuje některé komponenty (angl. „garbage-collector“), které dělají programování snazší a rychlejší. Současně byl zvolen i kvůli velmi propracovanému konektoru k databázi MySQL [4]. Programovací jazyk pro tvorbu funkcionality na klientovi (vytváření nových sekcí pro další požadavky, zobrazování a skrývání oblastí) byl zvolen jazyk Javascript [6] původně v jeho základní syntaktické verzi. Během vývoje jsem přešel na jeho nástavbu jQuery, která je výborně popsána na [9].

5.1 Programová struktura systému AVGMIS

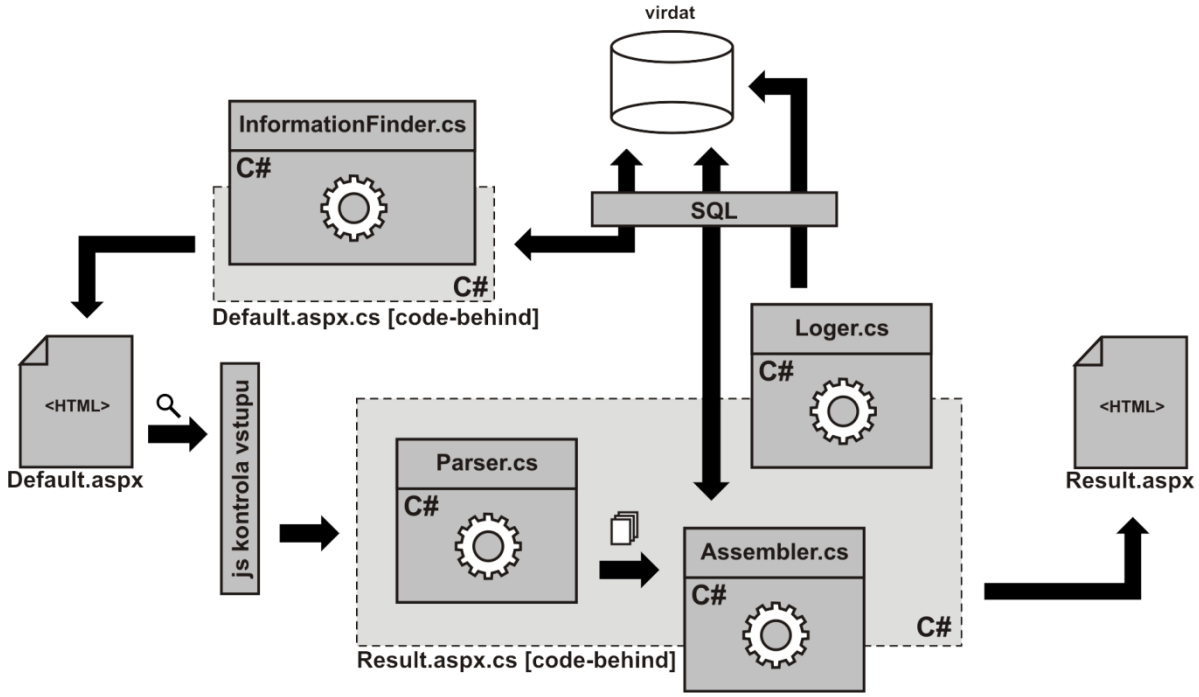
Tato kapitola se zbývá programovou strukturou systému AVGMIS. Na následujících řádcích tedy bude popsáno, jaké jsou v rámci sytému AVGMIS vytvořeny třídy a jak jsou objekty těchto tříd provázány mezi sebou. Dále, v následujících podkapitolách, bude popsáno, jakou funkčnost mají jednotlivé třídy v sobě implementovánu.

Systém AVGMIS se skládá z pěti hlavních tříd. Jedná se o třídy `Parser`, `Assembler`, `Loger`, `InformationFinder` a `MySQLConnector`. Grafické znázornění, jak jsou tyto třídy mezi sebou provázány, je k vidění níže v obrázku. Na obrázku současně vidíme, že celý systém sestává pouze ze dvou webových stránek – požadavku a výsledku.

První stránkou je `Default.aspx` – stránka požadavků. Tato stránka zprostředkovává uživateli vstupní rozhraní, jehož části byly popsány a vyobrazeny výše v rámci kapitol o návrhu (celé grafické rozhraní je poté k vidění v přílohách této diplomové práce). Z pěti výše zmíněných programových tříd je v rámci stránky `Default.aspx` využita pouze třída `InformationFinder`. Tato třída umožňuje načíst z databáze některé elementární hodnoty – slovníky (typy souborů, názvy výrobců antivirů, seznamy packerů a archivátorů, …) a nabízí je uživateli v rámci různých výběrů. Tím zvyšuje komfort uživatele při výběru atributů a minimalizuje potenciální chybu vstupu (při přepsání).

Druhou stránkou je `Result.aspx` – stránka výsledků –, která v rámci svého kódu zpracovávaného webovým serverem obsahuje právě onu aplikační logiku a po jejím vykonání

uživateli nabízí nalezené vzorky. Tato stránka používá třídy `Parser`, `Assembler` a `Loger`. Všechny tři tyto zásadní třídy jsou konkrétněji popsány dále. Plné grafické rozhraní je opět k vidění v přílohách.



Obrázek 47: Grafické znázornění implementace systému AVGMIS

Dříve než přejdeme k vysvětlení implementace tříd aplikační logiky, chtěl bych ještě alespoň ve stručnosti popsat chování aplikace jako celku vzhledem k implementaci. Vše začíná vygenerováním stránky `Default.aspx`, kde jsou uživateli nabídnuty možné vyhledávané atributy. Poté co uživatel vyplní formulář svými požadavky, odešle jej na server. V rámci javascriptového kódu se provede předběžná kontrola vstupu a pokud je tento vstup validní, je odeslán serveru na zpracování. Na serveru je přijat a postupně zpracován. Poté co jsou požadavky uloženy (parsovány), jsou identifikovány do skupin. Dále je z těchto skupin na základě požadavků určen scénář vyhledávání – neboli povaha skupiny. Poté co jsou tyto scénáře identifikovány, je podle návrhu popisovaného výše sestaven dotaz za danou skupinu. Po sestavení všech dotazů jsou dotazy prováděny kaskádovitě. Výsledné pole identifikátorů je poté zasláno do výsledkové stránky a je generován uživatelský výstup.

5.1.1 Parser

`Parser` je první ze tří hlavních tříd aplikační logiky uvnitř systému AVGMIS. Tato třída má na starosti správné rozebrání vstupu na jednotlivé hodnoty. Její funkcionalita spočívá v rozebírání jednotlivých vstupních rozhraní a jejich konverzi do interní datové struktury (o použitých datových strukturách se později zmiňují v podkapitole Zpracování programových datových struktur).

V rámci rozšířeného vstupního rozhraní („Advanced“) se jedná o výběr hodnot z předem definovaných vstupů v rámci formuláře. Třída musí samozřejmě pružně reagovat na skutečnost, že není zcela pevně dáno, kolik kterých polí je v požadavku obsaženo. Z povahy zadání přece plyne, že počet jednotlivých argumentů není omezen ve všech případech.

V případě vstupu příkazovou řádkou („Cmd“) se jedná o rozklad vstupního řetězce, který obsahuje jak hodnoty požadované/vyhledávané uživatelem, tak klíčová slova, která určují o jaké atributy se jedná (sémantické označení jednotlivých atributů). Tento rozklad se pak děje na základě regulárních výrazů a jejich následného zpracování.

Třída Parser tedy rozloží hodnoty vstupu do předem připravené struktury, aniž by věděla, o jaké hodnoty se jedná nebo jaký mají význam pro pozdější vyhledávání. Současně tato třída tvoří jakousi pomyslnou druhou úroveň pro kontrolu vstupu. Ve skutečnosti se jedná o kopii ochrany vstupu, kterou v první úrovni poskytuje javascript. Tato úroveň nevznikla jen jako potenciální pojistka, kdyby se podařilo obejít kontrolu javascriptu, ale jako potřeba pro kontrolu přímého vstupu („Cmd inline“), který je popisován v rámci návrhu. Tento vstup žádnou javascriptovou kontrolu neobsahuje, protože přistupuje přímo k výsledkové stránce a její aplikační logice.

5.1.2 Assembler

Pravděpodobně nejdůležitější třída v rámci aplikační logiky. Tato třída obdrží datovou strukturu, ve které jsou uloženy hodnoty vstupů. Poté, co tuto strukturu obdrží od objektu třídy Parser, musí objekt třídy Assembler identifikovat skupiny obsažené v požadavcích. Tato identifikace probíhá na základě implementované znalosti (návrh rozřazení konkrétních typů atributů do skupin je uveden výše). Jakmile jsou skupiny identifikovány (vzniknou pole atributů spadajících do dané skupiny a seřazené podle možné ovlivnitelnosti výsledků – viz popis řídící tabulky ctrl_avgmis), tak je dále identifikován typ scénáře pro jednotlivé skupiny. K tomu se opět použije interní datové struktury zaslané objektem třídy Parser.

V okamžiku, kdy je identifikace skupin a jejich scénářů dokončena, přistoupí objekt třídy Assembler k sestavení jednotlivých dotazů SQL. Po sestavení dotazů jsou postupně zasílány do databáze, přesně podle výše zmíněné hierarchie zpracování. Současně po každém dotazu je kontrolován počet identifikátorů v daném dočasném úložišti (počet výsledků po provedeném dotazu). Je to z toho důvodu, aby byl splněn požadavek na zkrácené vyhodnocení v případě, že by daný stupeň kaskády nic nevrátil. Poté co objekt Assembler sestaví a položí všechny dotazy databázi, vznikne ve výsledném úložišti (posledním stupni kaskády) výsledný seznam identifikátorů – nalezených vzorků. Tento výsledný seznam je současně výstupem z metody Assembly() v rámci objektu třídy Assembler.

5.1.3 Loger

Třída `Loger` je odpovědná za vytváření záznamu o probíhajícím vyhledávání. Podstata této třídy tedy spočívá ve sběru dat, měření délky provádění dotazů jednotlivých skupin a následném ukládání naměřených hodnot do databáze. Tato data jsou později použita pro analytické účely (analýze dat z jednotlivých hledání se dále věnuje kapitola Testování systému v reálném nasazení).

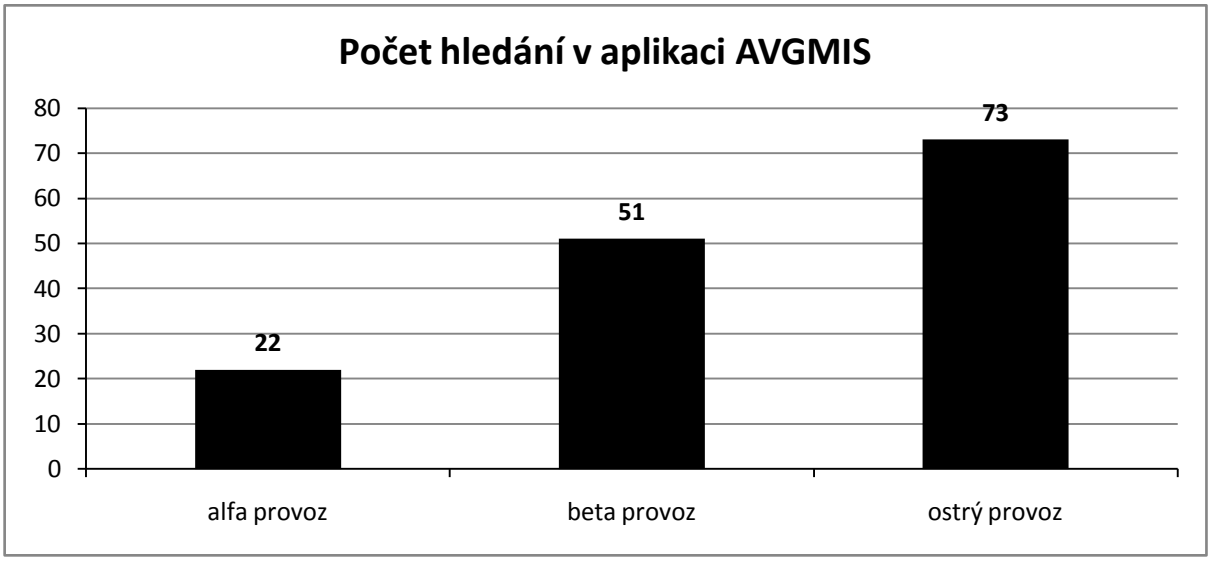
Jak je vidět, třída `Loger` svou funkčností nespadá přímo do aplikační logiky. To ale neubírá na její důležitosti. Ve skutečnosti tato třída tvoří jedinou cestu, jak získat přehled o jednotlivých prováděných hledáních. Tyto údaje posléze mohou sloužit jako vodítko k identifikaci pomalých dotazů. Současně se touto formou dají stanovit i priority, které dotazy optimalizovat, protože jsou požadované častěji než jiné.

Celý proces zaznamenávání je spustitelný a odpojitelný přes soubor `web.config`, konkrétně pak změnou hodnoty `true/false` ve značce s klíčem nazvaným „`logs`“. Tento způsob umožňuje ovládání zaznamenávání bez potřeby rekompilace zdrojových kódů (zdrojové kódy v jazyce C#, které jsou použity jako „code-behind“, je nezbytné při změnách v kódu znovu překompilovat).

6 Testování systému v reálném nasazení

Tato kapitola se zabývá testovacím provozem systému AVGMIS. Kromě popisu některých chronologických dat z testování jsou v podkapitolách níže popsány i informace o analytickém rozhraní. Tato sekce popisuje jak datové úložiště, kde jsou záznamy z hledání uloženy, tak i grafická rozhraní a jejich možnosti. V poslední podkapitole je vyhodnocení provedených hledání. Toto hodnocení má za úkol nastínit, k jakým hledáním se systém AVGMIS používá. Toto hodnocení může být později využito pro lepší optimalizaci konkrétních požadavků.

Testovací provoz začal 1. srpna 2011, kdy byla spuštěna vůbec první verze systému, která již obsahovala všechny položky výše zmíněného návrhu. Tato verze byla zpřístupněna pouze několika kolegům ze společnosti AVG Technologies a téměř okamžitě přinesla některé věcné připomínky. Tento „alfa provoz" skončil 20. září 2011, kdy bylo provedeno pouhých dvacet dva hledání. Od 21. září 2011 se rozběhl „beta provoz", který byl ve znamení širší prezentace produktu na oddělení virové analýzy. Od této doby se pak systém využíval více. Celkový počet hledání v rámci „beta provozu" byl padesát jedna. 25. listopadu 2011 byl systém prezentován v rámci přednášky o zpracovávání vzorků plénu složenému z oddělení virové laboratoře a kompletního oddělení vývoje. Prezentaci systému (zmínku o využitelnosti systému v rámci vyhledávání vzorků) provedl Pavel Krčma. V tuto chvíli již končí pomyslný „beta provoz" a systém je otevřen volnému užívání, ve kterém běží dodnes. V dnešnímu datu (8. února 2012) systém záznamů vyhledávání eviduje celkem sto čtyřicet šest hledání, což je od ukončení „beta provozu" právě sedmdesát tři hledání. Grafické znázornění počtu užití v jednotlivých obdobích je uveden níže.



Obrázek 48: Grafické znázornění počtu užití systému AVGMIS

6.1 AVGMIS – analytické rozhraní

Tato podkapitola se zabývá popisem analytického rozhraní systému AVGMIS. Podobně jako vlastní systém AVGMIS je toto rozhraní implementováno jako webová stránka. Opět zde existuje vstupní rozhraní, které umožňuje vyhledávat podle určitých atributů, a rozhraní výstupní, které zobrazuje nalezené záznamy o hledání. Současně je v následující podkapitole popis části databáze `virdat`, kde jsou ukládána data z hledání a mechanismy, jak tato data publikovat do výstupního rozhraní.

6.1.1 `log_avgmis` – úložiště záznamů

Tabulka `log_avgmis` slouží jako úložiště jednotlivých záznamů o hledání. Celá tabulka by se dala rozdělit na pět pomyslných oblastí. První čtyři oblasti slouží k zaznamenávání informací o hledáních nad jednotlivými skupinami (skupina strukturálních indexů, skupina detekcí, skupina files a skupina umělého plnění).

Každá z těch čtyř oblastí má pět sloupců. První sloupec (`q_??`) nese podobu vygenerovaného dotazu SQL. Druhý sloupec (`t_??`) obsahuje čas v milisekundách, který značí, za jak dlouho se daná sekce vykonala. Třetí sloupec (`cnt_??`) obsahuje počet identifikátorů vrácený skupinou. Čtvrtý sloupec (`order_??`) potom značí pořadí, na kolikátém místě se daná skupina zpracovávala. Poslední pátý sloupec (`search_??`) je odkazem (identifikátorem) do slovníku scénářů a v podstatě tedy říká, v rámci jakého scénáře byla skupina zpracovávána.

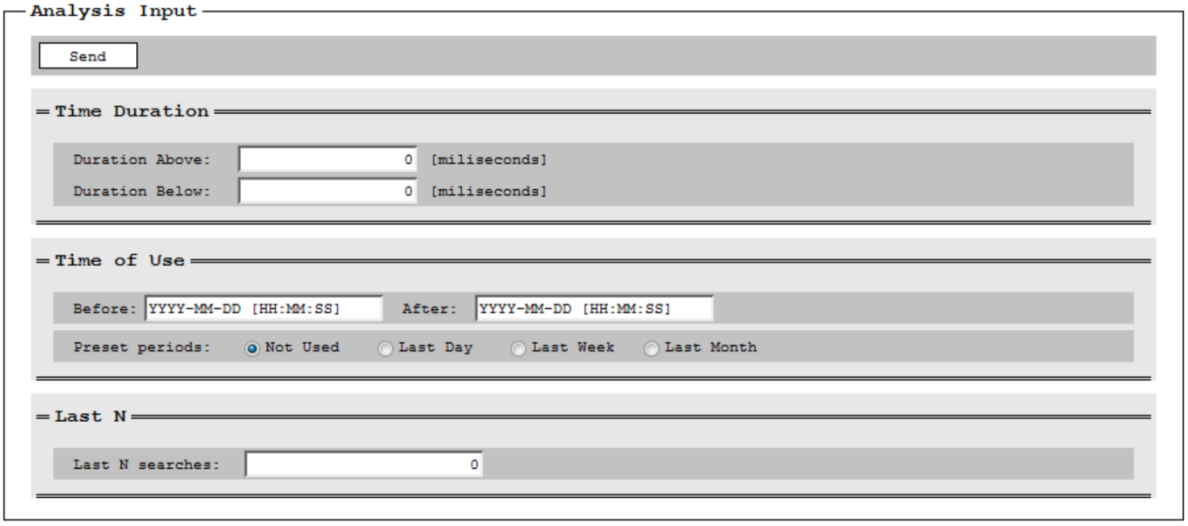
Poslední úsek tabulky `log_avgmis` obsahuje sloupce, které uvádějí globální informace pro každý jeden záznam. Jedné se o informaci, kdy byl záznam pořízen (`used`), identifikátor záznamu (`id_log`), dobu, za jak dlouho byl celý požadavek proveden (`t_all`) – doba za, kterou byly provedeny všechny požadované skupiny, a konečně celý řetězec „`command string`“ (`cmd_inline`), který umožňuje rychlou analýzu požadavku bez potřeby zkoumání vlastních dotazů.

Na konci této podkapitoly bych se ještě zmínil o způsobu, jakým jsou data z databáze publikována do analytického rozhraní. Podobně jako v systému AVGMIS se nakonec data nestahují běžným dotazem SQL, ale byl na ně vytvořen tzv. databázový pohled (angl. `view`), konkrétně pak `view_log_avgmis`. Podstata tohoto databázového pohledu spočívá v jednotné formě, s jakou jsou data z databáze stahována. Tento pohled, jehož podoba je uvedena níže, čerpá prostřednictvím několika vnořených dotazů ze slovníku scénářů. Současně jsou v pohledu definovány uživatelsky přijatelnější názvy jednotlivých sloupců, čímž pohled přispívá k mnohem jednoduššímu programovému zpracování.

6.1.2 Vstup

Vstup analytického rozhraní byl pojat ve stejném grafickém duchu jako u systému AVGMIS. Opět se zde jedná o relativné jednoduchou grafiku, která velmi přehledně nabízí možné atributy vyhledávání. V případě analytického rozhraní jsou uživateli nabídnuty tři atributy.

Prvním z těchto atributů je délka zpracování dotazu. Tento typ vyhledávání se může hodit, pokud se snažíme analyzovat dlouho trvající dotazy. Dotaz je položen podle sloupce `t_all` (viz podkapitola `log_avgmis` – úložiště záznamů). Druhým možným způsobem je vyhledávání záznamů podle času (sloupec `used`), což umožňuje zaměřit se na určité časové období. Tyto varianty je možno kombinovat mezi sebou. Třetí a poslední variantou je vyhledávání posledních N záznamů (hledání v aplikaci AVGMIS). Tento přístup je určen pro potřeby analýzy využití systému, o které se zmiňuje jedna z následujících podkapitol. Současně tento atribut není možné kombinovat s předešlými dvěma.

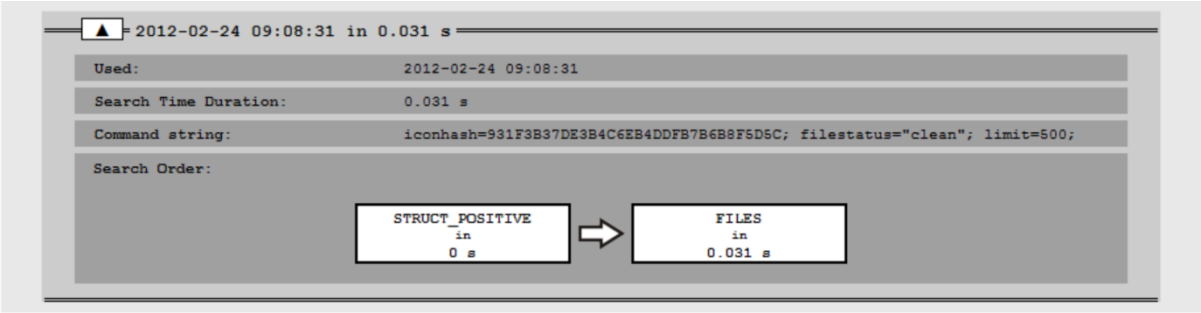


Obrázek 49: Grafické znázornění vstupu analytického rozhraní systému AVGMIS

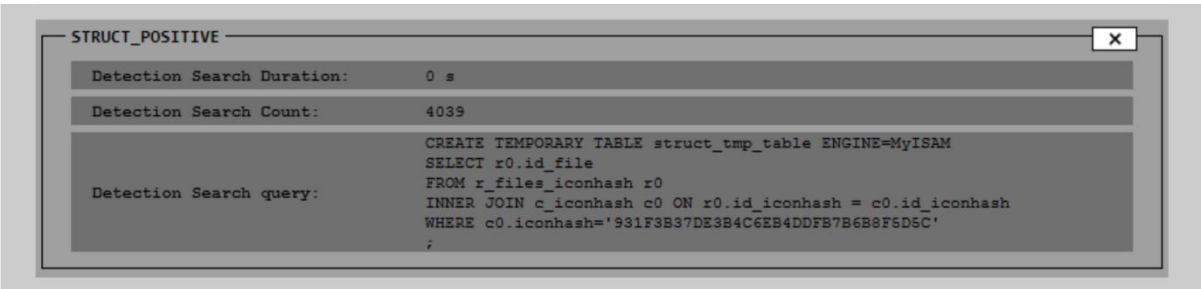
6.1.3 Výstup

Výstup analytického rozhraní (vyobrazený níže) je rozdělen opět do tří různých úrovní. První úroveň je přehledová, tedy uvádí pouze, kdy byl záznam proveden a jak dlouho trval. Druhá úroveň již ukazuje také to, co bylo hledáno. Tedy je zde publikován řetězec „`command string`“, aby bylo možno provést toto hledání znovu (metoda přímého vstupu do systému AVGMIS pomocí řetězce „`command string`“). V druhé úrovni je také možno vidět sekci, která popisuje pořadí zpracování skupin (`search_order`). Ve třetí a poslední úrovni jsou již uvedeny jednotlivé detailní informace v rámci každé vyhledávací skupiny.

Obrázek 50: Grafické znázornění výstupu analytického rozhraní – první úroveň



Obrázek 51: Grafické znázornění výstupu analytického rozhraní – druhá úroveň



Obrázek 52: Grafické znázornění výstupu analytického rozhraní – část třetí úrovně

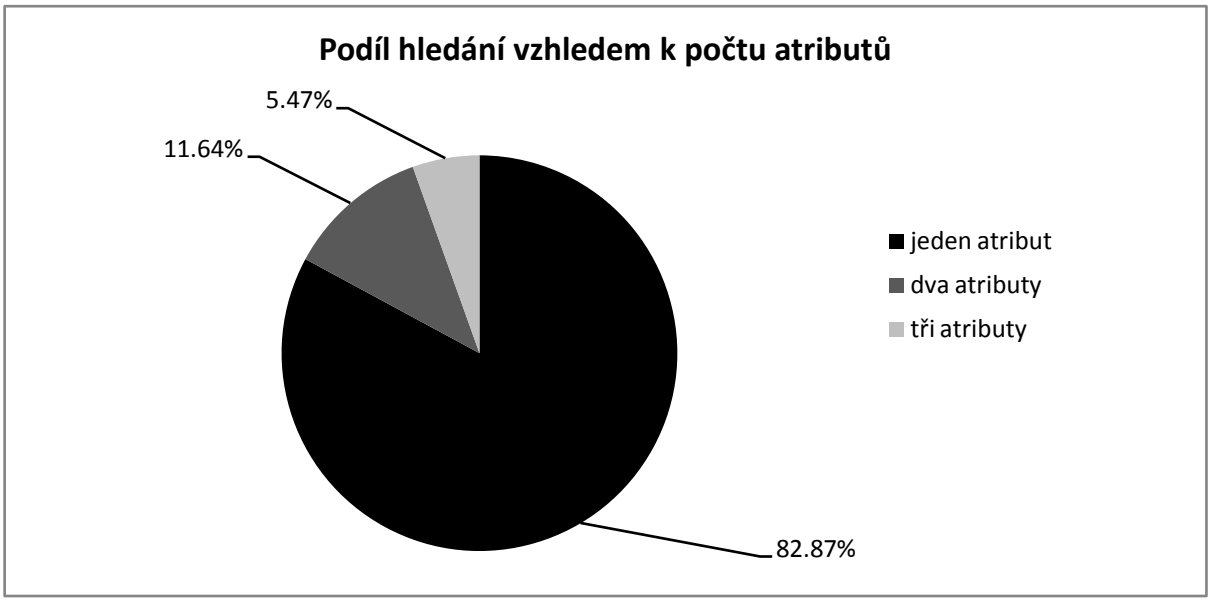
6.2 Hodnocení využití systému AVGMIS během testovacího provozu

Tato kapitola se zabývá analýzou hledání, která byla za dosavadní dobu života systému AVGMIS provedena. Následující řádky tedy budou vykreslovat do jisté míry povahu uživatelů – jejich požadavků. Současně zde bude dobře vidět, jak komplexní požadavky do systému přicházely.

Jako zdrojová množina hledání byla použita ona výše zmíněná skupina sto čtyřicet šesti požadavků. Analýza se bude zaměřovat na počet atributů v požadavku (z tohoto pohledu se dá odhadnout jistá míra komplexnosti) a dále pak na určité typy atributů v požadavcích.

Počty atributů

Co se týče počtu atributů, objevily se v systému za celou jeho existenci tři druhy hledání. Největším zástupcem jsou hledání, která mají pouze jeden atribut. Takových požadavků bylo celkem sto dvacet jedna, což tvoří něco málo pod osmdesát tři procent všech hledání (82,87 %). Dalším zástupcem jsou požadavky se dvěma atributy, kterých v systému proběhlo sedmnáct, což odpovídá zhruba jedenácti procentům všech hledání (11,64 %). Zbývajících osm požadavků patří hledáním se třemi atributy. To odpovídá pouze pěti procentům z celkových sto čtyřiceti šesti hledání (5,47 %). Grafické znázornění těchto hledání je uvedeno níže.



Obrázek 53: Grafické znázornění podílu hledání podle počtu atributů

Různorodost vyhledávaných atributů

Zajímavou částí hodnocení provedených hledání je jejich povaha, neboli jaké typy atributů jsou uživateli hledány. Z drtivé většiny převažuje hledání vzorků určitých detekcí. Tato hledání jsou zastoupena v celkem sto sedmi případech. V některých z těchto případů bylo použito i vícenásobné hledání detekcí. V osmdesáti devíti případech byly vyhledávány přímo detekce podle společnosti AVG Technologies. Mezi další atributy, které byly systémem ještě vyhledávány, jsou `secthash`, hash z certifikátu, typ souboru a `packer`.

Z výše provedeného rozboru plyne, že jsou vyhledávány vzorky převážně na základě názvů detekcí, zejména pak detekcí společností AVG Technologies. Ostatní atributy jsou zastoupeny v celkem malé míře.

Implementace slovníku

V kapitole Zkrácené vyhodnocení a Limitace velikosti výsledku je krátký odstavec, který se zabývá potenciálními problémy, které mohou vzniknout při velkém objemu vzorku v prvním stupni kaskády. Podle vyhodnocení různorodosti a hlavně počtu atributů, které jsou po systému požadovány, bylo stanoveno, že implementace slovníku nebude provedena. Sice tím ponecháme toto potenciální riziko nekryté, ale vzhledem k formám užívání to zatím efektivitu hledání neovlivní.

Závěr hodnocení

Shrneme-li hodnocení požadavků na hledání systémem AVGMIS, pak vidíme, že systém je momentálně používán pouze v jeho nejjednodušší formě. Slouží především jako vyhledávač vzorků, které jsou detekovány společností AVG Technologies. Rozhodně se dá říct, že uživatelé systém zatím nevyužívají v síle, v jaké byl navrhován. Nadějí je vzestupná četnost hledání, která je vyobrazena na

grafickém znázornění na konci podkapitoly Testování systému v reálném nasazení. Ta naznačuje, že s postupem času se využití systému bude postupně zvedat.

Reakce kolegů ze společnosti AVG Technologies naznačují, že nástroj je z jejich pohledu určen pro velmi specifická hledání, kdy je třeba najít jistou množinu vzorků, které se určitým způsobem podobají. Tohle je jedna z možností, jak systém využívat. Druhou možností je nabídnout vzorky uživatelům neznalým jazyka SQL.

7 Hodnocení optimalizací

Tato kapitola ukazuje výsledky testů, které byly provedeny za účelem podpory rozhodování týkajícího se výběru efektivních variant konstrukcí v jazyce SQL. Obvykle jsou v rámci každého testu uvedeny dvě varianty konstrukce v jazyce SQL z niž jedna je efektivnější než druhá. Kompletní podobu testu (jsou připojeny i konkrétní podoby konstrukcí v jazyce SQL) jsem umístil do příloh. Je tomu z důvodu úspory prostoru v těle práce.

V rámci každého testu jsou v podkapitolách níže uvedeny jejich výsledky (čas dotazu a jeho počet provedení za minutu) a dále jsou k nim připojeny komentáře, které vystihují jejich podstatu. Současně u některých z nich je uvedeno, jestli jsou implementovány, popřípadě v jaké formě tomu tak je.

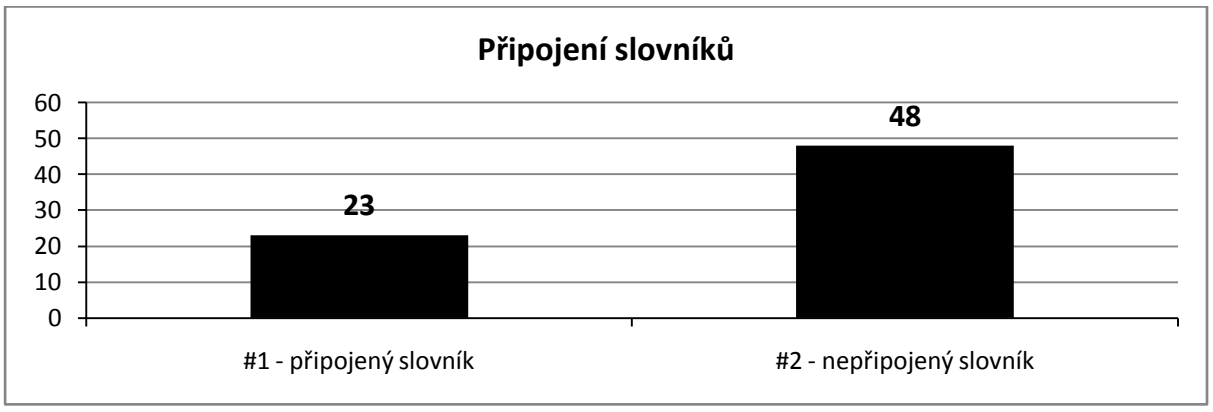
Testů se týkají dvě podkapitoly. První z nich popisuje obecné optimalizace, tedy ty, které nejsou vázány na navrhovanou a implementovanou aplikaci a jednoduše dokazují či vyvrací tvrzení z publikace [1]. Druhá podkapitola už ukazuje výsledky testů algoritmů, které byly implementovány v aplikaci.

7.1 Obecné optimalizace

Tato podkapitola se snaží dokázat či vyvrátit tvrzení z publikace [1]. Současně výsledky těchto testů byly použity jako vodítko pro později implementované optimalizace. Dále se tyto optimalizace dají považovat za vůbec základní kameny v oblasti podmínkové části dotazu SQL.

Připojení slovníků

Podstata této optimalizace spočívá v nepřipojování slovníku vyhledávaných hodnot (celé tabulky nesoucí význam identifikátorů) do hlavního dotazu SQL. Jednoduše se jedná o snahu slovník vypsat do uživatelského rozhraní a aplikační logice tak poskytnout jen identifikátor do tohoto slovníku, který obvykle bývá na pravé straně vazební tabulky, která má za úkol slovník připojit k bázové tabulce.

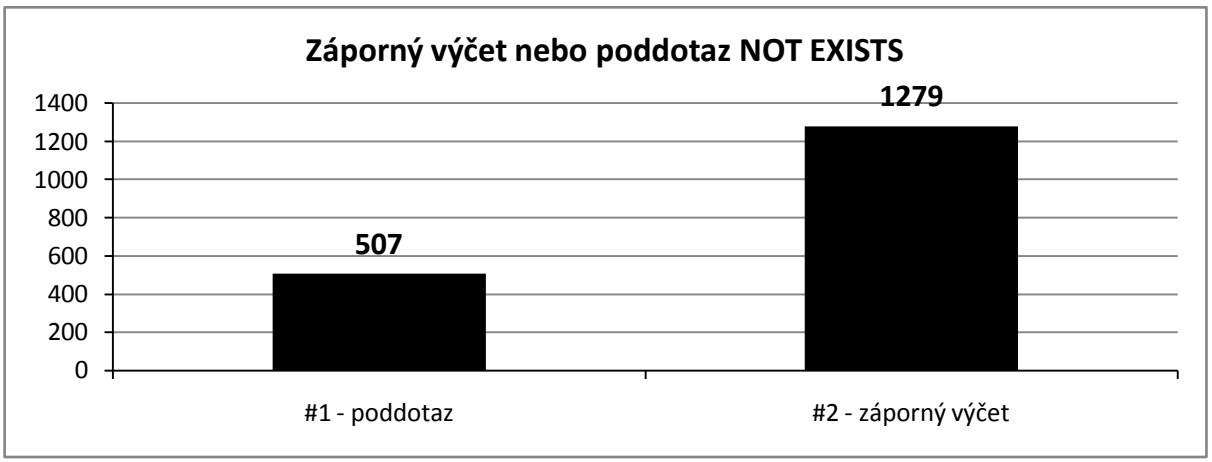


Obrázek 54: Grafické znázornění výkonu testu (soubor `1A.sql`)

Test celkem bez problému prokázal, že připojení slovníku zpomalí dotaz o polovinu. Tento rozdíl se může ještě více prohloubit, pokud by byl dotaz spuštěn na celé tabulce `files`. Jelikož je výsledek dostatečně výřečný, postačil pro tento test pouze milion vyhledaných vzorků.

Záporný výčet nebo poddotaz `NOT EXISTS`

Tento test má za úkol ukázat, jestli při hledání více „záporných" hodnot je vhodnější použít výčet nebo poddotaz `NOT EXISTS`. Současně je nutné zdůraznit, že požadujeme vzorky, které nemají ani jeden ze zadaných atributů.



Obrázek 55: Grafické znázornění výkonu testu (soubor `1B.sql`)

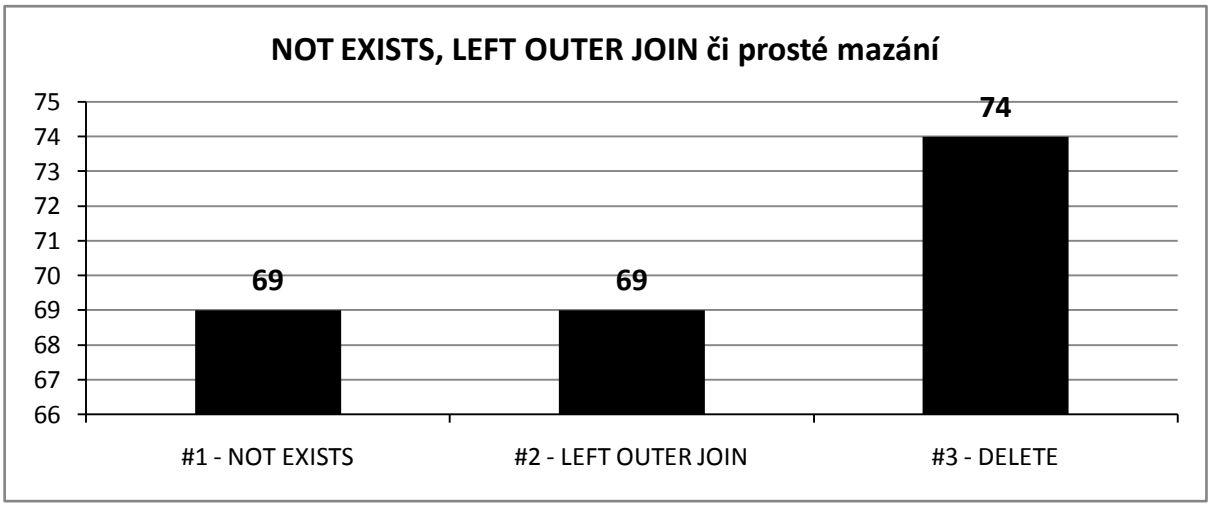
Test ukázal, že varianta výčtu je rychlejší než při použití klauzule `NOT EXISTS`. Ostatně i autoři publikace [1] popisují zkušenosti s klauzulemi `NOT EXISTS` jako nepříliš příznivé a často je doporučují přepsat na jinou ekvivalentní variantu. Podmiňují to ovšem provedením testu, aby se toto tvrzení potvrdilo.

U tohoto testu bych rád zmínil návaznost na aplikaci AVGMIS. Je totiž velmi pravděpodobné, že by tento způsob mohl v některé z dalších verzí aplikace nahradit stávající algoritmus odmazávání. Tyto a další změny jsou popsány v kapitole o dalších možných vylepšeních.

`NOT EXISTS, LEFT OUTER JOIN` či prosté mazání

Tento test je následníkem předchozího testu. Autor publikace [1] popisuje klauzuli `LEFT OUTER JOIN` jako možnou náhradu za `NOT EXISTS`. Současně jsem přidal ještě možnost prostého odmazání z dočasného úložiště kvůli výše zmíněné programové podobnosti s kladnými požadavky na atributy (kladné scénáře provádění dotazů výše v textu).

V příloze je vidět ještě rozdílnost limitů. Je tomu z toho důvodu, aby bylo zhruba dosaženo stejné velikosti výsledku.

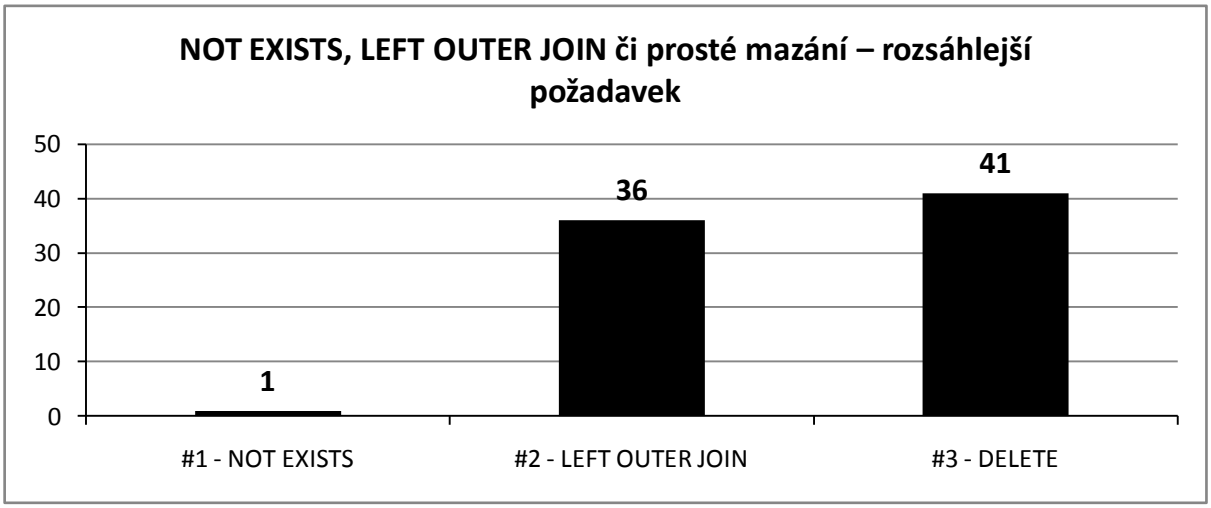


Obrázek 56: Grafické znázornění výkonu testu (soubor `1C.sql`)

Výsledek tohoto testu má velký význam pro aplikaci AVGMIS. Tento dotaz ukazuje, jak se „zbavit" vzorků, které jsou „zabalené". Byť klauzule `DELETE` nenabízí velký náskok, oproti ostatním dvěma klauzulím je pořád rychlejší.

`NOT EXISTS, LEFT OUTER JOIN` či prosté mazání – rozsáhlejší požadavek

Tento test je obdobou předchozího testu a ukazuje, jak se databáze MySQL vypořádá s požadavkem, kdy hledáme vzorky, které nemají záznamy v určitých dvou tabulkách. Tento typ dotazu je využíván, stejně jako v předchozím případě, hledáme-li vzorky, které nejsou „zapakovány", ale teď nově hledáme i takové, které nejsou „archivovány".



Obrázek 57: Grafické znázornění výkonu testu (soubor `1D.sql`)

Výsledky testu opět prokázaly, že klauzule `DELETE` je rychlejší než ostatní dvě. Zajímavé je vidět, jak se zvýšil čas na vykonání poddotazu s klauzulí `NOT EXISTS`. Ovšem výsledek klauzule `LEFT OUTER JOIN` je také velmi ucházející, proto by mohla být považována za potenciální budoucí

obměnu klauzule `DELETE` aktuálně implementovanou v aplikací AVGMIS. Výhoda klauzule `LEFT OUTER JOIN` spočívá v jednodušší integraci do kladného dotazu.

7.2 Optimalizace použité v aplikaci AVGMIS

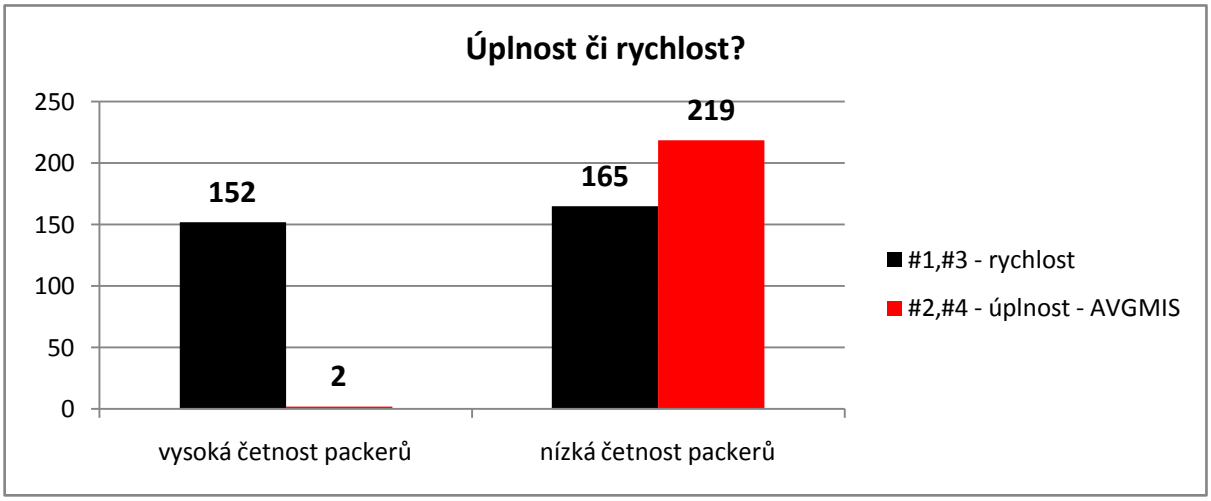
Tato podkapitola ukazuje výsledky testů z implementovaných algoritmů v aplikaci AVGMIS. Je rozdělena na dvě menší podkapitoly, které se zabývají specifickými skupinami. První z nich je skupina strukturálních indexů a skupina `files`, druhou skupinou je skupina detekcí, která si svou komplexností (už při návrhu) vydobyla svou vlastní podkapitolu.

Ještě než přejdeme na následující podkapitoly, budou zde popsány dva testy, které se přímo dotýkají implementace aplikace AVGMIS. První z nich je test ukazující problematiku rychlosti a úplnosti na reálném požadavku. Druhý pak ukazuje, jak se reálně projeví připojení slovníku při skutečném aplikačním požadavku, a nejen v „laboratorních podmínkách“, jak tomu bylo v předešlé podkapitole.

Úplnost či rychlost?

Tento test se dotýká problematiky mezi úplností a rychlostí, která je popisována v podkapitole Zkrácené vyhodnocení a Limitace velikosti výsledku. Důvodem, proč je zde vůbec zahrnut, je ukázka reality, kterou nám tyto dva přístupy mohou poskytnout.

V testu jsou ukázány čtyři dotazy (konstrukce v jazyce SQL jsou uvedeny v přílohách). Každý lichý dotaz využívá ořezávání prvního stupně kaskády, tedy kloní se spíše k požadavkům na rychlost. A každý sudý dotaz zase ukazuje realitu při požadavku na úplnost. První dva dotazy vybírají v prvním stupni kaskády vzorky, kterých je k dnešnímu datu (17. 2. 2012) něco přes jedenáct milionů. Tento stupeň je omezen na padesát tisíc vzorků. V druhém případě (dotazy `#3` a `#4`) se jedná o vzorky, které jsou v databázi zastoupeny podstatně méně (je jich méně než padesát tisíc). Tím pádem není omezení vlastně třeba.

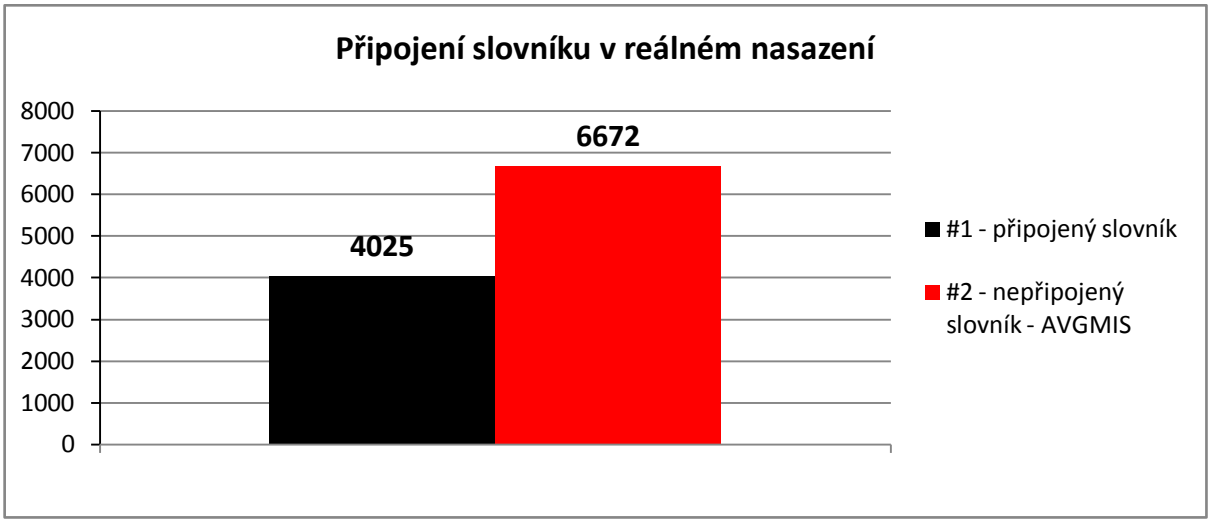


Obrázek 58: Grafické znázornění výkonu testu (soubor `2G.sql`)

Z výsledků je vidět, že omezení intervalu v prvním stupni kaskády má smysl jedině tehdy, pokud je dat výrazně více, než je tento interval (v případě packeru `UPX` je to až dvě stě dvacetkrát více). Současně, požadujeme-li jen tento malý díl z databáze (padesát tisíc vzorků), pak se výrazně zvýší pravděpodobnost minutí výsledku. Dalším aspektem, proč jsem nakonec od ořezání prvního stupně kaskády upustil, je fakt, že tento nástroj dle testů využití není užívám k vyhledávání příliš velkých obecností, ale spíše naopak, takže je požadována spíše úplnost než rychlost.

Připojení slovníku v reálném nasazení

Tento test ukazuje, jak se změní výkon dotazu, pokud je k němu připojen malý slovník či pokud hledáme pouze identifikátor této slovníkové položky. Do jisté míry se testy z předchozí podkapitoly podobají, ale v tomto případě přibyl ještě požadavek na konkrétní hash z ikon (`iconhash`), takže už nejde jen o „umělý dotaz“, ale o ukázku reálného použití implementovanou v aplikaci AVGMIS.



Obrázek 59: Grafické znázornění výkonu testu (soubor `2A.sql`)

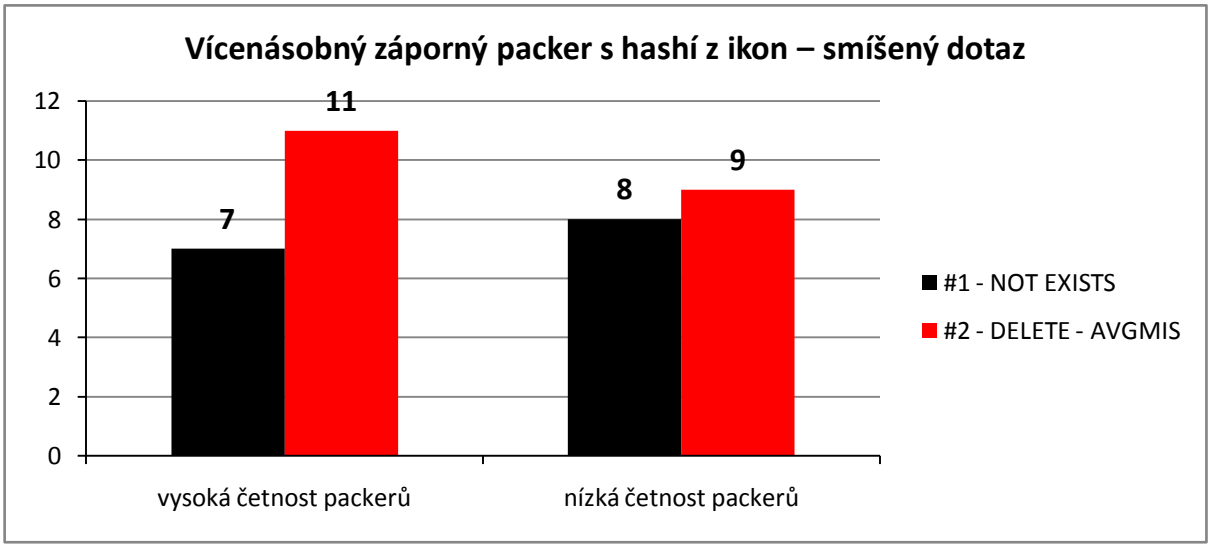
Výsledek testu dopadl dle očekávání: velmi podobně jako v prvním případě. Jen tentokrát je vidět, že rozdíl není už tak propastný, jako tomu bylo v předchozím případě. Je to z toho důvodu, že celková množina požadovaných hashí z ikon není v takovém množství, jako je množina dat se statusem `infected` (hodnota dva ve sloupci `STATUS`), a současně je tato množina menší než požadovaný limit. Podstatným poznatkem na tomto testu zůstává, že nepřipojování malých slovníků má velký význam na výkon dotazu.

7.2.1 Skupiny strukturálních indexů a `files`

Tato podkapitola nabízí výsledky testů implementovaných algoritmů pro skupinu strukturálních indexů a skupinu `files`. K vidění budou testy ukazující smíšené scénáře skupiny strukturálních indexů, výkon dotazů s globálním záporem a ukázku kaskády s omezením intervalu prvního stupně.

Vícenásobný záporný `packer` s hashí z ikon – smíšený dotaz

Tento test ukazuje výkonnost aplikace AVGMIS v případě, že požadujeme určitou `iconhash` a současně sní (dotazy `#1` a `#2`) i vzorky, které nejsou zabaleny packery `UPX` a packery `unknown` (neidentifikovaný packer). V druhém případě (dotazy `#3` a `#4`) se pak jedná o stejnou `iconhash`, ale o jiné dva packery – `Yoda` a `Molebox` – (s výrazně menším zastoupením vzorků v databázi).



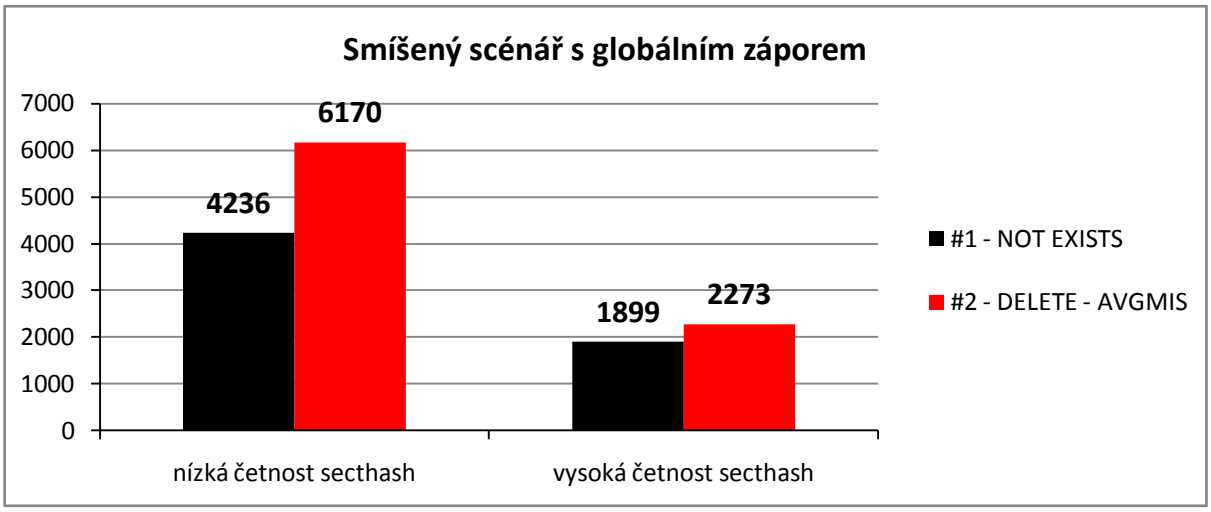
Obrázek 60: Grafické znázornění výkonu testu (soubor `2B.sql`)

Test ukázal, že varianta implementovaná v aplikaci AVGMIS je stále efektivnější (klauzule `DELETE FROM`), než konstrukce SQL tvořené poddotazy. V případě dotazů (`#3` a `#4`) je vidět, že pokud mají packery menší zastoupení (menší počet vzorků) konstrukce SQL v aplikaci AVGMIS pomalu ztrácí na efektivitě.

Smíšený scénář s globálním záporem

V tomto dotazu je testována situace, kdy požadujeme konkrétní hash ze sekcí (`secthash`) a certifikát a současně chceme, aby tyto vzorky nebyly vůbec zabalené (globální zápor). V testu jsou opět testovány dva scénáře, které ukazují, jaký vliv na výkon mají rozdílná množství výskytů vzorků.

V prvním případě (konstrukce `#1` a `#2`) se jedná o `secthash` s relativně malým výskytem (sto třicet devět vzorků) a v druhém případě (dotazy `#3` a `#4`) se jedná o `secthash` s vyšším výskytem (2262 vzorků). V podstatě se jedná o velmi podobný dotaz jako v předchozím případě s tím rozdílem, že teď se mění četnost v záporné části dotazu a pak je použit globál zápor (odstranit všechny vzorky, co existují v jedné tabulce).



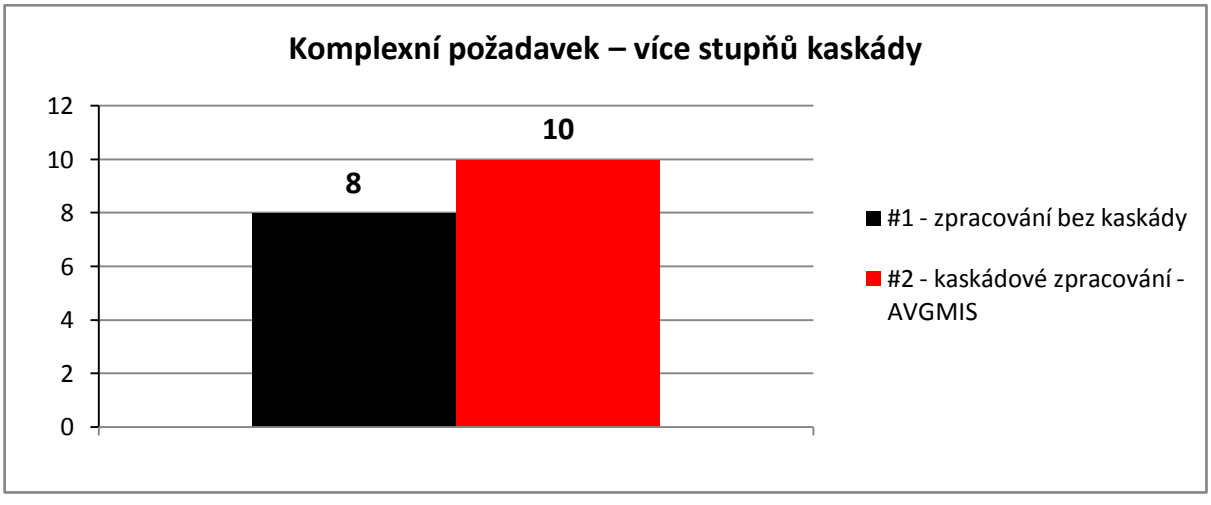
Obrázek 61: Grafické znázornění výkonu testu (soubor `2C.sql`)

Výsledky testů opět mluví ve prospěch dotazů implementovaných v aplikaci AVGMIS. Současně zde máme možnost vidět zpomalení druhé sady dotazů v případě, že množina dat získaných z kladné části je větší než v prvním případě. Poté bude trvat déle odmazání dat ve fázi aplikace globálního záporu. Přesto je tato varianta stále rychlejší.

Komplexní požadavek – více stupňů kaskády

Tento test se řadí mezi jedny z nejkomplexnějších testů, které jsem v rámci testování výkonu na aplikaci AVGMIS prováděl. Jedná se o relativně různorodý požadavek, kdy hledáme infikované vzorky zabalené archivátory `NSIS`, `CAB`, `ZIP` a přebalené packerem `UPX`. Současně jako třešničku na dortu hledáme ty vzorky, které mají v sekci `resources`, konkrétně pak v položce „`company name`“, řetězec Microsoft.

Pokud provedeme stručný rozbor tohoto požadavku, pak zjistíme, že se ve skutečnosti jedná pouze o kladný scénář, byť na dvou stupních kaskády. Nakonec ještě tento výsledek omezíme limitem na sto takových vzorků.



Obrázek 62: Grafické znázornění výkonu testu (soubor `2F.sql`)

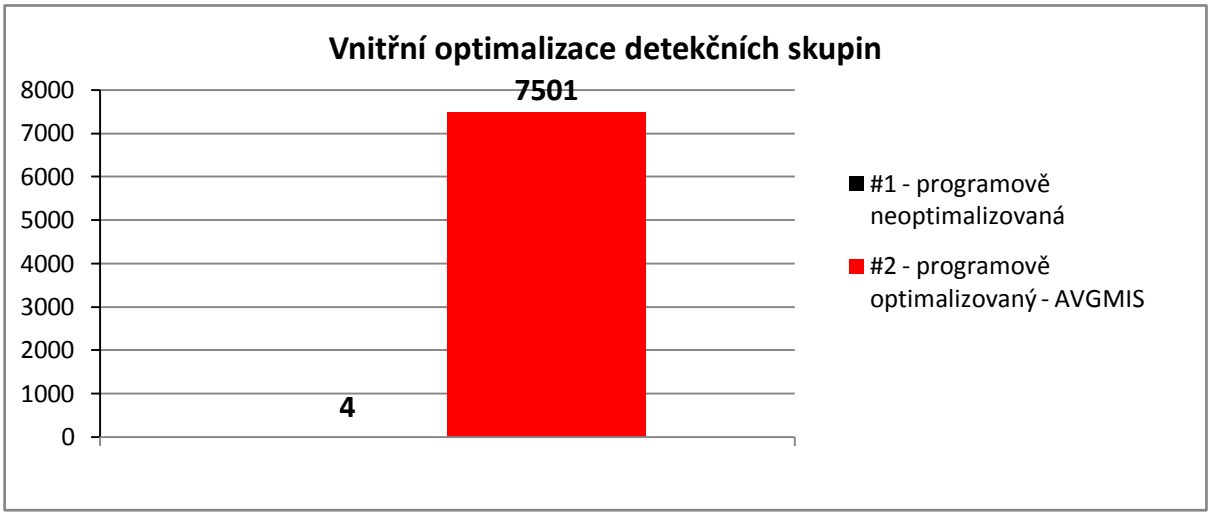
Výsledky, kterých dosáhlo implementované řešení, jsou o něco lepší než v případě klasického jednoho velkého dotazu. Tento rozdíl ovšem není příliš výrazný a vzhledem k tomu, jak je dotaz rozsáhlý (velké množství podmínek), je provedení jednolitého dotazu databázovým strojem velmi dobré. Tento výsledek, ač dopadl pro aplikaci AVGMIS ještě slušně, vnáší jistou pochybnost o potřebě kaskády. Zvláště pokud by se velikost požadovaných vzorků výrazně zvětšila, mohlo by dojít k minutí výsledku.

7.2.2 Skupina detekcí

Tato podkapitola nabízí dva testy, které se zabývají hledáním ve skupině detekcí. První z testů ukazuje kromě efektivnějšího dotazu do databáze taky provedení vnitřní optimalizace, která je implementovaná v aplikaci. Druhý test pak ukazuje, jak je na tom řešení aplikace AVGMIS v případě požadavku na tři různé detekční skupiny. V takovém případě není možné provést vestavěnou optimalizaci.

Vnitřní optimalizace detekčních skupin

Tento test ukazuje, jaký vliv má vnitřní optimalizace v aplikaci AVGMIS na výsledný dotaz, který se později zasílá databázi. Optimalizace spočívá v kontrole vstupu a hledání podobností. Zrovna v tomto příkladu je ukázán případ, kdy uživatel požaduje vzorky, které jsou detekovány jako „`Adware%`“, nebo pak požaduje druhou skupinu vzorků, které jsou detekovány jako „`Adware%`“, ale nejsou detekovány antivirem Kaspersky. Jelikož v kapitole požadavků na četnosti atributů je stanoveno, že detekční skupiny mezi sebou mají logický vztah OR, a nadále se v tomto případě dá ještě tvrdit, že druhá požadovaná skupina detekcí je podmnožinou té první, pak program druhou detekční skupinu vypustí a provede jen tu první, čímž ušetří jeden dotaz.



Obrázek 63: Grafické znázornění výkonu testu (soubor `3A.sql`)

Z výsledků testu je vidět, že se tato programová optimalizace velmi hodila. Rychlost zpracování je v porovnání s neoptimalizovanou verzí rychlejší v řádu několika tisíců. Samozřejmě ve skutečnosti nejde o optimalizaci vlastního dotazu, ale pouze o optimalizaci požadavku. V následujícím testu je ukázána efektivní metoda, jak pokládat dotazy na detekční skupiny bez programové podpory.

Vícenásobné detekční skupiny

Tento test je posledním, který je v diplomové práci popisován. Konkrétně se zabývá srovnáním výkonu při požadavku na vzorky, které jsou detekovány firmou AVG jako „`%Agent.6.Au%`“ a nejsou detekovány firmou McAfee. V druhé skupině požadujeme vzorky detekované opět společností AVG jako „`%VBS/Downloader.agent%`“ a opět nejsou detekovány společností McAfee. Konečně ve třetí detekční skupině požadujeme vzorky detekované společností AVG jako „`Trojan Horse%`“ a nechceme aby byly detekovány společností Avira. Jak je vidět, je tento požadavek dostatečně různorodý a tudíž neumožňuje žádné optimalizace založené na podobnosti.

Obrázek 64: Grafické znázornění výkonu testu (soubor `3B.sql`)

Výsledek testu uvedený výše ukazuje na jisté obtíže optimalizace klauzule `UNION`. Řešení aplikované v systému AVGMIS je v tomto případě padesátkrát rychlejší než ekvivalentní dotaz s klauzulí `UNION`. Test byl proveden opakovaně a dodával stejné výsledky. Současně se dá předpokládat, že klauzule `UNION` bude do budoucna optimalizována vývojovým týmem MySQL, neboť klauzule není v paletě příkazů dlouho. Tento typ dotazu (s klauzulí `UNION`) byl použit, protože velmi přehledně znázorňuje požadavek, který byl položen. Proto je možné, že tento dotaz by použil méně zběhlý uživatel právě pro jeho názornost.

8 Možná další vylepšení

Tuto kapitolu jsem rozdělil na tři části. První část se týká možných vylepšení databázového schématu, druhá část se zabývá vylepšeními nad algoritmy, které jsou v systému AVGMIS implementovány, a třetí část se zabývá rozšířeními samotné aplikace pro další použití.

Vylepšení databázového schématu

Po stránce databázového schématu existuje skutečně jen málo vylepšení, která mě ve spojení s aplikací AVGMIS napadají. Celé databázové schéma je výsledkem několikaleté práce kolegů z oddělení virové analýzy. Důkaz o tom, že se jedná o velmi kvalitní práci, je k vidění uvnitř testů (rychlosti zpracování dotazů SQL), které jsou publikovány výše. Dalším důkazem budiž skutečnost, že celý databázový stroj je server MySQL v komunitní verzi, a i přesto je schopen více než dobře zpracovávat náročné dotazy při poměrně velkých objemech dat.

Jedna z mála věcí, které by se mohli v rámci aplikace vylepšit, je indexování. Zejména mám na mysli sloupce `VERINFO_???`, které obsahují řetězcové hodnoty a jejich vyhledání bez indexu je zdlouhavé. Avšak z hodnocení užití systému je vidět, že tato hledání se prakticky neprovádí, takže indexování těchto sloupců asi nebude v nejbližší době na pořadu dne.

Další aspekt, který by mohl výrazně zrychlit vyhledávání, je paměť databázového serveru. V tuto chvíli není celá databáze načtena do paměti a určitá její část se nachází stále na disku. Pokud by se jí podařilo do paměti načíst celou (do budoucna se počítá s výrazným navýšením do paměti), pak by se mohl výkon zvýšit.

Vylepšení aplikace AVGMIS

V této kapitole bych rád uvedl některá potenciální vylepšení, která by měla umožnit širší využití aplikace AVGMIS. Původní záměr vystavět webové rozhraní (databázový „frontend“) nad databází `virdat` (původní aplikace AVGMIS) se celkem ujal a byl implementován dle výše zmíněných skutečností. Ovšem padly i další návrhy, jak vylepšit celou aplikaci.

Jedním z těchto návrhů je uživateli nabídnout stažení vzorků z naší databáze. K tomu byl Pavlem Šimonem aplikován systém Norman Share Sample server, který umožňuje stáhnout nalezené vzorky z databáze. Toto rozšíření bylo implementováno do aplikace. Současně s tímto rozhraním bylo třeba implementovat i dekodér těchto stažených souborů (soubory jsou na cestě ze serveru šifrovány). Dekodér jsem úspěšně implementoval a služba stažení vzorků aplikací AVGMIS funguje a je používána.

Dalším vylepšením, jak využít aplikační jádro (optimalizace hledání), je částečné omezení grafického rozhraní a tím pádem potenciální nasazení do konzolových aplikací, které hledají vzorky podle podobností. Jistá odlehčená forma již existuje ve formě přímého hledání popisovaného výše

v textu. Úplné zproštění od grafického rozhraní, které se potenciálně bude implementovat, by umožnilo plné využití aplikačního jádra. Předběžný návrh v takové případě uvažuje, že aplikace bude řízena pouze řetězcem „`command string`“ a výsledkem bude pouze seznam hashí `MD5` či jiná textová forma, která při správném rozložení (angl. parsing) poskytne seznam vzorků.

Vylepšení vyhledávacích algoritmů

Zde, téměř na konci této práce, je třeba se kriticky zamyslet i nad implementovanými algoritmy a jejich budoucností. Část hodnocení je umístěna v testech, ale ty neříkají nic o tom, co s nepříznivými výsledky, které se sem tam začínají objevovat. Během posledních měsíců se velikost virové databáze významně zvýšila. V době, kdy byly prováděny první návrhy algoritmů na vyhledávání, měla databáze něco málo pod třicet milionů záznamů (jak dokazuje původní podoba tabulky `ctrl_avgmis` uvedená v přílohách). Dnes je to přes čtyřicet pět milionů záznamů v bázové tabulce, která zaznamenává pouze unikátní vzorky. Vazební tabulky, které zaznamenávají vzorky modifikované (balené různými packery, archivátory a další) se zvýšily také významně. Konkrétně u packerů je z osmnácti milionů třicet a půl milionu. Z toho důvodu je efektivita zpracovávání algoritmy menší.

Do budoucna se jeví jako nejvíce přijatelná varianta změna kaskády na verzi, která se bude více podobat jednorázovému zpracovávání (jeden velký dotaz). Dále se jeví jako nejpravděpodobnější úplné odstranění prováděcích scénářů. Záporné zpracovávání mimo hlavní dotaz se ukazuje nadále jako neúčinné (již teď existují testy, které dokazují, že jednolité zpracovávání je výkonnější), protože odmazávání tak velkého počtu vzorků z dočasných tabulek se stává náročnou operací.

9 Závěr

Na závěr diplomové práce bych rád zhodnotil její přínos, jak praktický, tak teoretický. Co práce přinesla mně a co přinesla uživatelům, kteří využívají ve formě implementované aplikace. Na konec závěru bych ještě rád uvedl některá číselná data, která jistým způsobem vystihují rozsáhlost aplikace.

Po praktické stránce se jedná o funkční rozhraní k virové databázi. Jelikož je systém prakticky nasazen a používán, dá se říci, že má praktický přínos. Současně umožňuje získat data i uživatelům neznalým jazyka SQL. Také umožňuje bezpečný přístup partnerům společnosti AVG ke vzorkům, které kdy do firmy dorazily.

Po teoretické stránce si dovoluji tvrdit, že i zde je přínos. Práce ukazuje, jaké jsou možnosti virové databáze a jaký je výkon použitých dotazů. Současně tato práce ukazuje některé vhodné postupy, jak provádět optimalizace na databázových dotazech nezávisle na virové databázi. Efektivitu některých aplikovaných optimalizací dokazuje provedenými testy. Současně naznačuje i některé slepé cesty v důsledku zvyšující se velikosti databáze.

Na otázku, co tato práce přinesla mně, musím odpovědět: jednoznačně zkušenosti. A nebylo jich málo. Poznatky, že ne každý dotaz, který vrací stejnou množinu dat, se provádí stejně a je stejně efektivní. Poznatky z tvorby grafického uživatelského rozhraní, které se během vývoje aplikace několikrát změnilo. Dále to bylo několik poznatků během návrhu a implementace, kdy v případě výměny aplikačního jádra bude třeba provádět jen minimum změn.

Přínos pro uživatele, kteří aplikaci používají, spočívá v jednodušší práci s virovou databází, kdy již nemusí vymýšlet dotazy, ale prostě mohou použít nástroj. Současně doufám, že přínos (jednoduchost a logičnost) pro uživatele plyne i z práce v uživatelsky přívětivém rozhraní, které jsem s jejich připomínkami složil.

AVGMIS v číslech

- 291,9 kB kódu ve zdrojových souborech, které se zpracovávají na serveru (angl. „`code-behind`“), 82,5 kB javascriptu pro uživatelskou funkcionalitu webových stránek.
- Práce započaly studiem literatury 10. 6. 2010.
- Verze alfa byla vypuštěna dne 1. 8. 2011, následovaná verzí beta dne 21. 9. 2011 a zakončená plným provozem dne 25. 11. 2011.
- 141 různých druhů dotazů v jazyce SQL vedoucích k výslednému návrhu struktur SQL používaných v aplikaci AVGMIS, 30 různých dotazů v jazyce SQL při hodnocení výkonu popisovaných v této práci.

10 Literatura

[1] Schwartz, B., Zaitsev, P., Tkachenko, V., Zawodny, J., D., Lentz, A., Balling, D., J.: MySQL profesionálně: optimalizace pro vysoký výkon. Vyd. 1. Brno: Zoner Press, 2009, 712 s. ISBN 978-80-7413-035-9.

[2] MySQL 5.1 Reference manual: Optimizing SELECT Statements. [online]. [cit. 2012-03-15]. Dostupné z: http://dev.mysql.com/doc/refman/5.1/en/select-optimization.html

[3] MySQL 5.1 Reference manual: Storage Engines. [online]. [cit. 2012-03-15]. Dostupné z: http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html

[4] MySQL 5.0 Reference manual: MySQL Connector/NET. [online]. [cit. 2012-03-15]. Dostupné z: http://dev.mysql.com/doc/refman/5.0/es/connector-net.html

[5] Kofler, M.: Mistrovství v MySQL 5. Vyd. 1. Překlad Jan Svoboda, Ondřej Baše, Jaroslav Černý. Brno: Computer Press, 2007, 805 s. ISBN 978-80-251-1502-2.

[6] Flanagan, D.: JavaScript: kompletní průvodce. Vyd. 1. Překlad Jan Svoboda, Ondřej Baše, Jaroslav Černý. Praha: Computer Press, 2002, 825 s. ISBN 80-722-6626-8.

[7] Croft, J., Llyod, I., Rubin, D.: Mistrovství v CSS: pokročilé techniky pro webové designéry a vývojáře. Vyd. 1. Překlad Josef Bábík. Brno: Computer Press, 2007, 409 s. ISBN 978-80-251-1705-7.

[8] MSDN: C# Reference. [online]. [cit. 2012-03-15]. Dostupné z: http://msdn.microsoft.com/en-us/library/618ayhy6.aspx

[9] JQuery: JavaScript Library. [online]. [cit. 2012-03-15]. Dostupné z: http://docs.jquery.com/Main_Page

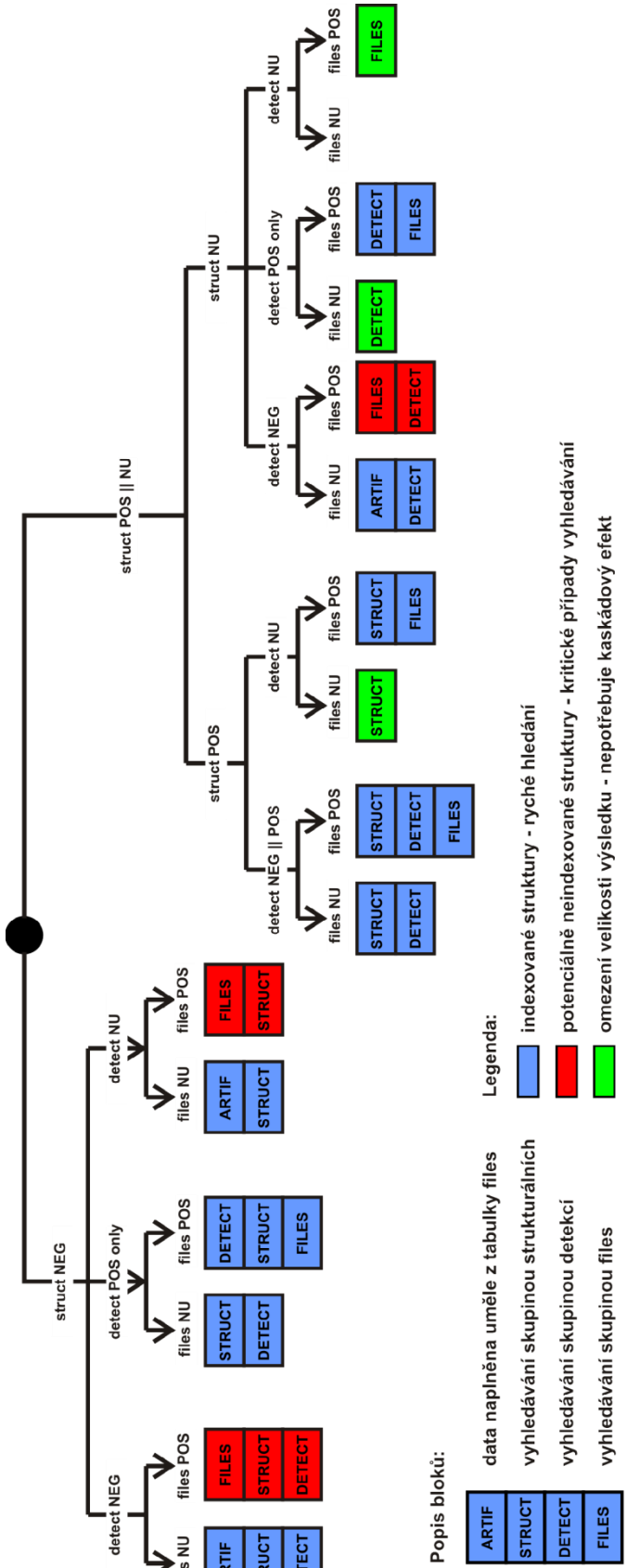
11 Příloha A

Tato příloha obsahuje obrázky znázorněné nebo odkazované v textu diplomové práce. Předností těchto obrázku je především větší rozlišení nebo úplnost. Obě tyto vlastnosti jsou problematicky splnitelné ve výše uvedeném textu, a proto jsou obrázky uvedeny právě zde.

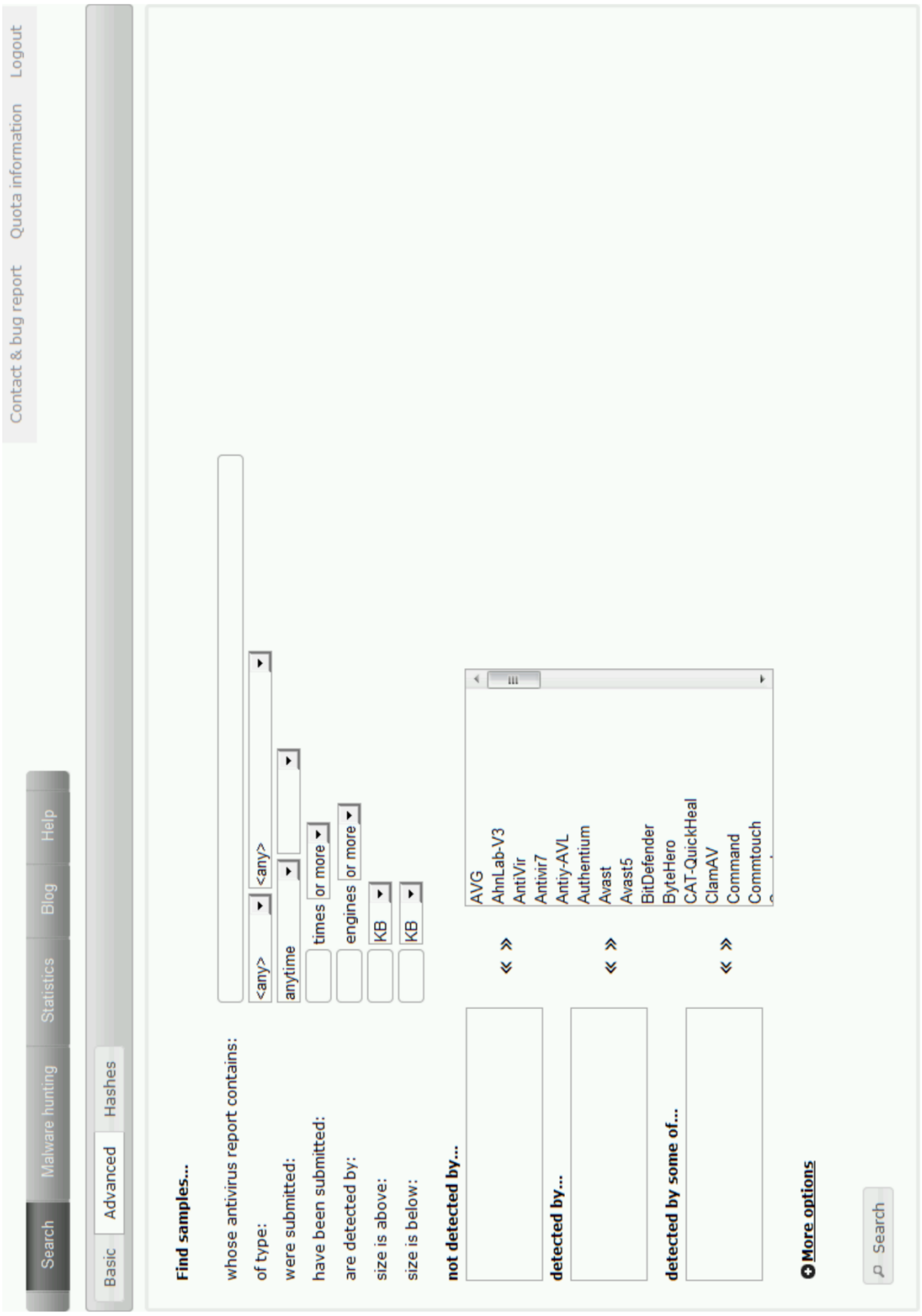
Tabulka ctrl_avgmis

section	index_type	relation	rec_count	r_table	r_join_attrib	c_table	c_join_attrib	c_where
certificate	0	0	555892	r_files_certificates	id_certificate	c_certificates	id_certificate	SHA256
group	0	1	1919625	r_files_groups	id_group	c_groups	id_group	id_group
archiver	0	1	3615944	r_file_archiver	id_archiver	c_archivers	id_archiver	archiver
det_ms	1	0	3978745	r_files_det_ms	id_det_ms	c_det_ms	id_det_ms	det_ms
filedesc	3	0	6993719	NULL	NULL	files	NULL	VERINFO_FileDescription
companyname	3	0	7071805	NULL	NULL	files	NULL	VERINFO_CompanyName
legalcopyright	3	0	7082557	NULL	NULL	files	NULL	VERINFO_LegalCopyright
fileversion	3	0	8949581	NULL	NULL	files	NULL	VERINFO_FileVersion
iconhash	0	0	11312275	r_files_iconhash	id_iconhash	c_iconhash	id_iconhash	iconhash
det_nvcc	1	0	13011697	r_files_det_nvcc	id_det_nvcc	c_det_nvcc	id_det_nvcc	det_nvcc
det_nod	1	0	13329915	r_files_det_nod	id_det_nod	c_det_nod	id_det_nod	det_nod
det_kav	1	1	14754660	r_files_det_kav	id_det_kav	c_det_kav	id_det_kav	det_kav
det_avira	1	0	14992945	r_files_det_avira	id_det_avira	c_det_avira	id_det_avira	det_avira
det_mcafee	1	0	15753097	r_files_det_mcafee	id_det_mcafee	c_det_mcafee	id_det_mcafee	det_mcafee
packer	0	1	18470781	r_file_packer	id_packer	c_packers	id_packer	packer
secthash	2	0	21698224	NULL	NULL	files	NULL	SECTHASH
det_avg	1	1	21741924	r_files_det_avg	id_det_avg	c_det_avg	id_det_avg	det_avg
filetype	3	0	29131043	NULL	NULL	files	NULL	TYPE
first_seen	3	0	29131043	NULL	NULL	files	NULL	FIRST_SEEN
filesubtype	3	0	29131043	NULL	NULL	files	NULL	SUBTYPE
filestatus	2	0	29131043	NULL	NULL	files	NULL	STATUS
filesize	3	0	29131043	NULL	NULL	files	NULL	FILESIZE

Kompletní hierarchie zpracovávání



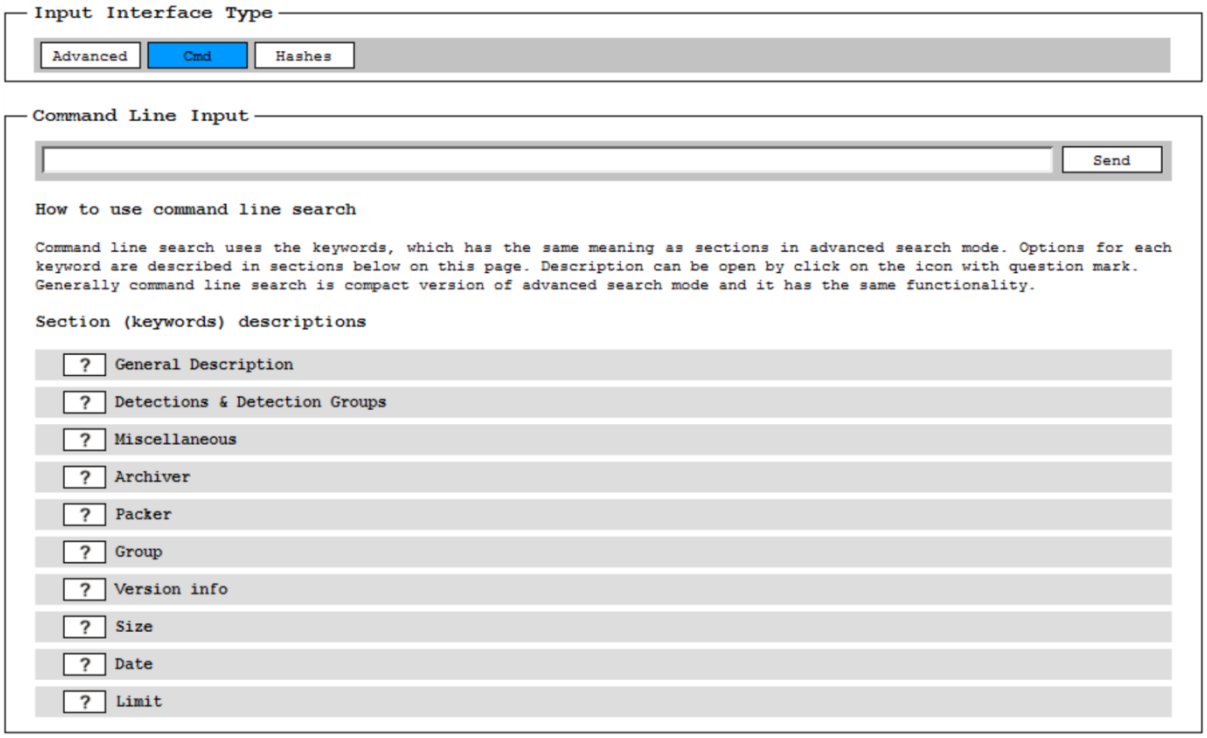
Aplikace VTMIS – rozšířený vstup



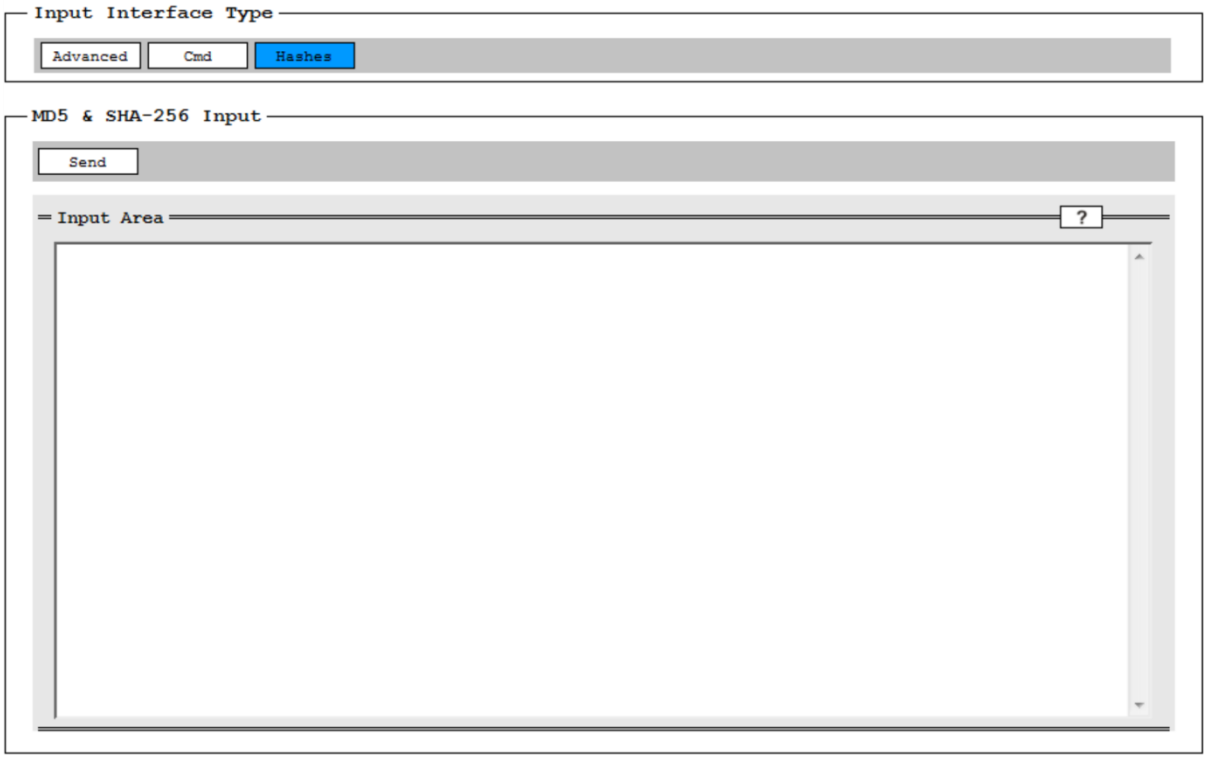
12 Příloha B

Tato příloha nabízí čtenáři pohled na implementovaná grafická uživatelská rozhraní v aplikaci AVGMIS. Vzhledem k jejich velikosti nebylo možné tyto obrázky vložit do textu práce.

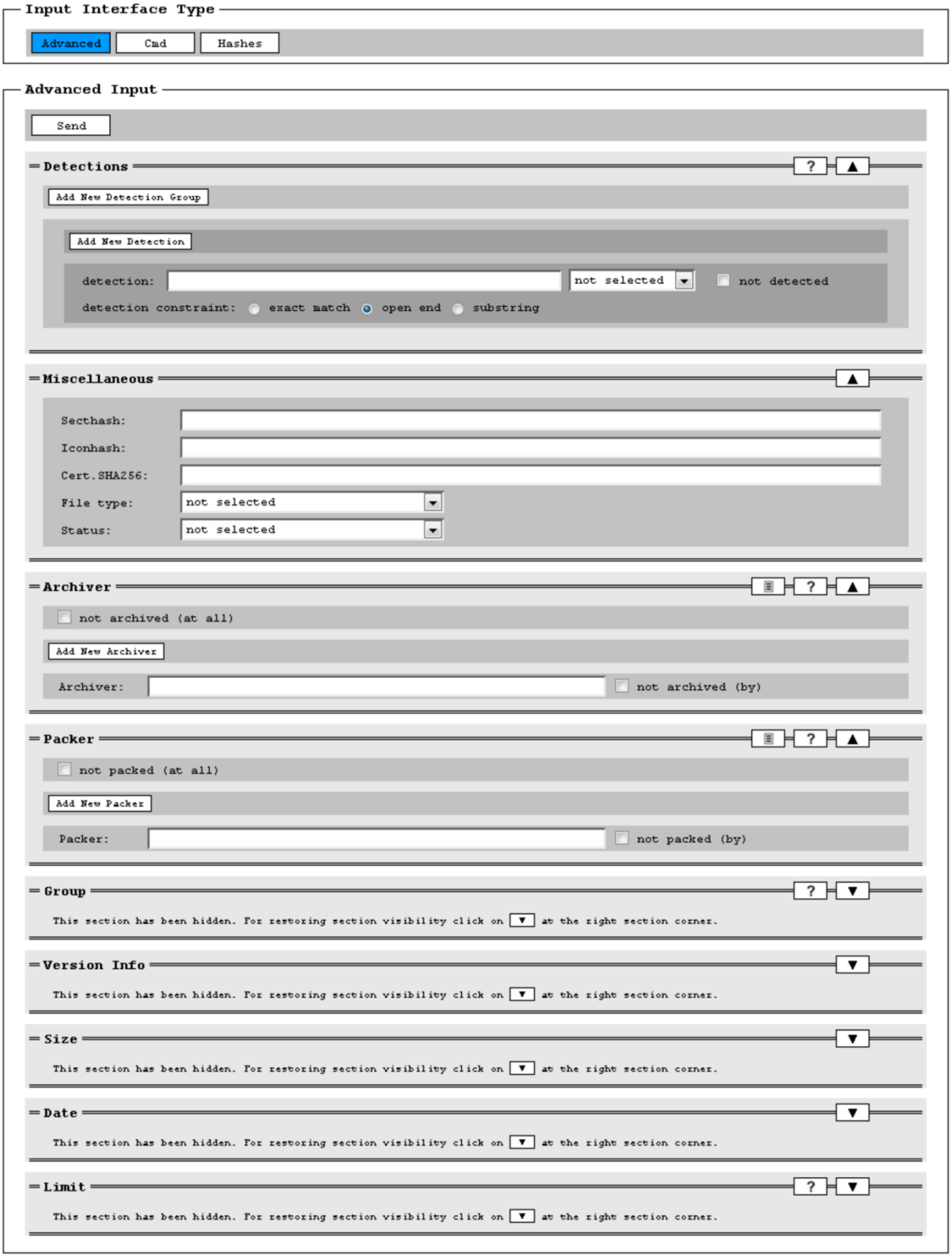
Rozhraní příkazové řádky aplikace AVGMIS



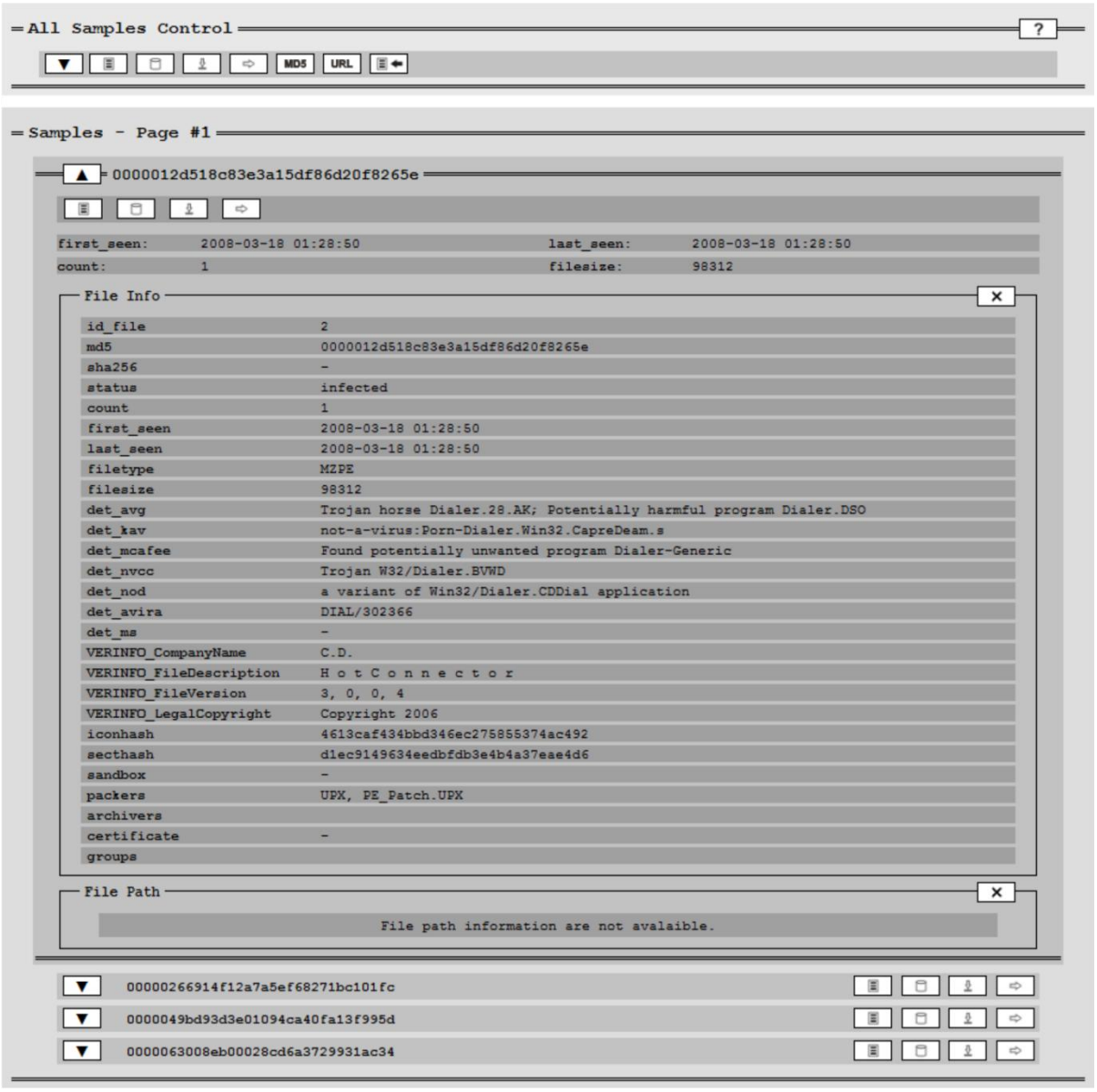
Rozhraní „Hashes“ aplikace AVGMIS



Rozhraní „Advanced“ aplikace AVGMIS



Výstup systému AVGMIS



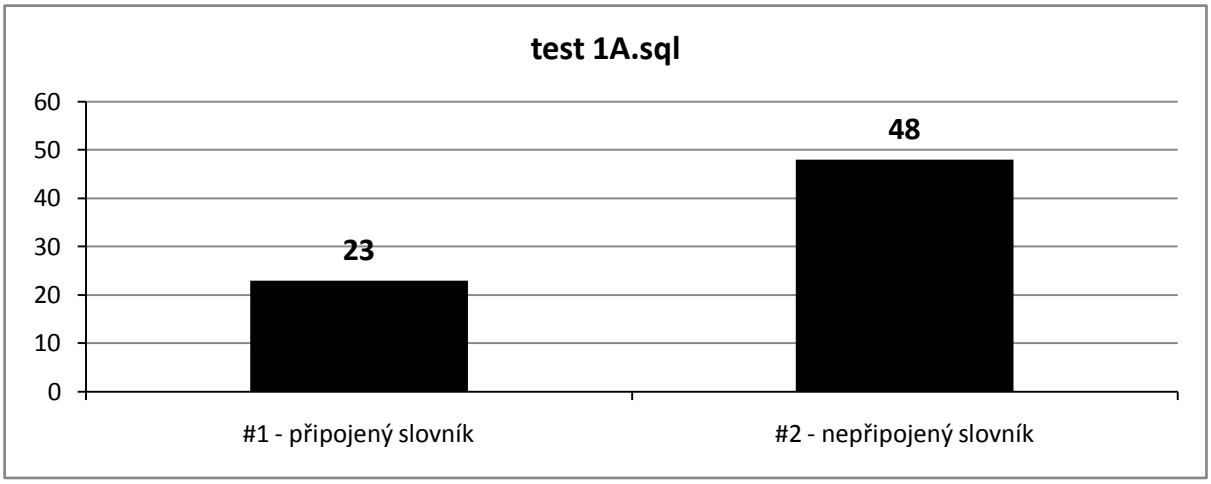
13 Příloha C

V této příloze najde čtenář přehledné a unifikované zobrazení jednotlivých testů, které byly v rámci testování výkonnosti aplikace AVGMIS provedeny. Každý test má určitou strukturu, v níž je uvedena podoba dotazu SQL a výsledek testu ve formě grafu (hodnota QPM). Konkrétní testovací skripty jsou uloženy na datovém nosiči, který je přiložen na zadní straně pevné vazby této diplomové práce.

Test 1A.sql

```
#1 - připojený slovník

SELECT SQL_NO_CACHE f.id_file
        FROM files f
        INNER JOIN c_file_status c ON f.STATUS = f.status
        WHERE
                c.meaning = 'infected'
        LIMIT 1000000;
```

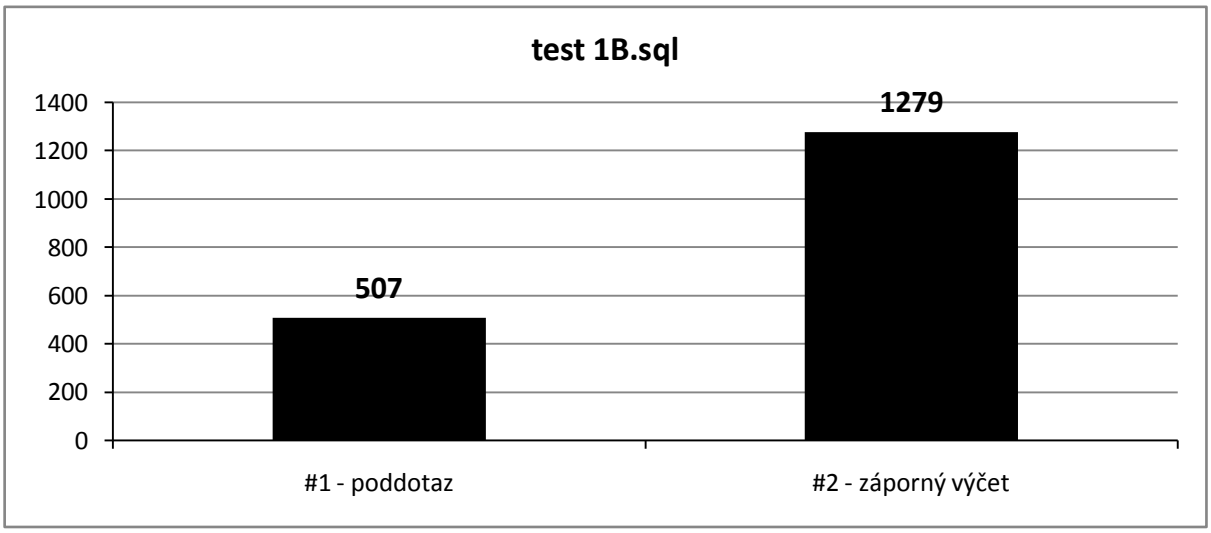
```
#2 - nepřipojený slovník

SELECT SQL_NO_CACHE f.id_file
        FROM files f
        WHERE
                f.STATUS = 2
        LIMIT 1000000;
```



Test 1B.sql

```
#1 - poddotaz

SELECT SQL_NO_CACHE f.id_file
        FROM files f
        WHERE
                NOT EXISTS(
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_archiver r
                                INNER JOIN c_archivers c ON r.id_archiver = c.id_archiver
                        WHERE
                                c.archiver = 'AutoIt' AND
                                f.id_file = r.id_file
                ) AND
                NOT EXISTS(
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_archiver r
                                INNER JOIN c_archivers c ON r.id_archiver = c.id_archiver
                        WHERE
                                c.archiver = 'Batch2Exe' AND
                                f.id_file = r.id_file
                )
        LIMIT 10000;
```

```
#2 - záporný výčet

CREATE TEMPORARY TABLE struct_tmp_table ENGINE=Memory
        SELECT SQL_NO_CACHE f.id_file
                FROM files f
                INNER JOIN r_file_archiver r ON f.id_file = r.id_file
                INNER JOIN c_archivers c ON c.id_archiver = r.id_archiver
                WHERE
                        c.archiver NOT IN('AutoIt','Batch2Exe' )
        LIMIT 10000;

DROP TEMPORARY TABLE struct_tmp_table;
```

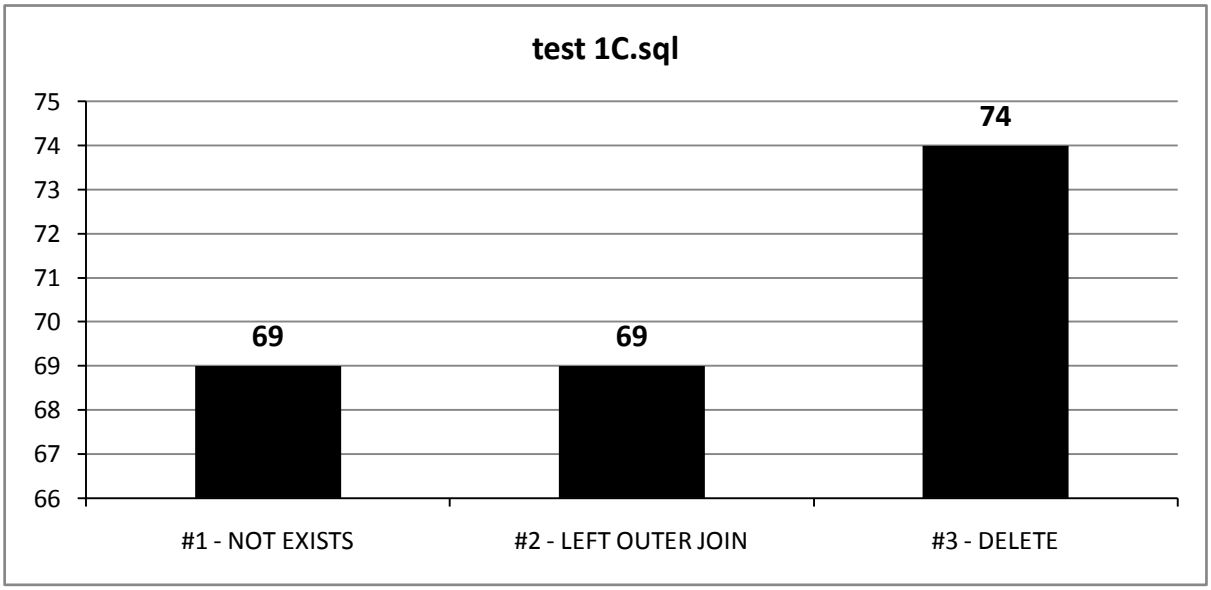


Test 1C.sql

```
#1 - NOT EXISTS

SELECT SQL_NO_CACHE f.id_file
        FROM files f
        WHERE
                NOT EXISTS(
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_packer r
                                WHERE
                                        r.id_file = f.id_file
                )
        LIMIT 100000;
```

```
#2 - LEFT OUTER JOIN

CREATE TEMPORARY TABLE struct_tmp_table ENGINE=MEMORY
        SELECT SQL_NO_CACHE f.id_file
                FROM files f
                LEFT OUTER JOIN r_file_packer r ON f.id_file = r.id_file
                WHERE
                        r.id_file IS NULL
        LIMIT 100000;

DROP TEMPORARY TABLE struct_tmp_table;
```

```
#3 - DELETE FROM

CREATE TEMPORARY TABLE files_tmp_table ENGINE=MEMORY
        SELECT SQL_NO_CACHE f.id_file
                FROM files f
        LIMIT 140000;

DELETE tmp
        FROM files_tmp_table tmp
        INNER JOIN r_file_packer r ON tmp.id_file = r.id_file
        WHERE
                tmp.id_file = r.id_file;

DROP TEMPORARY TABLE files_tmp_table;
```



Test 1D.sql

```
#1 - NOT EXISTS
```

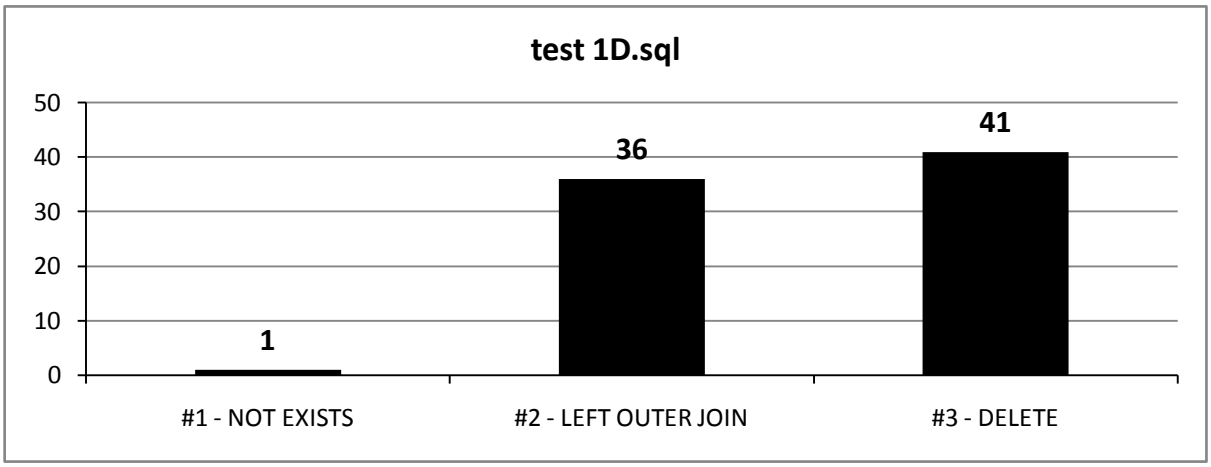
```
SELECT SQL_NO_CACHE f.id_file
        FROM files f
        WHERE
                NOT EXISTS(
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_packer r
                                WHERE
                                        r.id_file = f.id_file
                ) AND
                NOT EXISTS(
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_archiver r
                                WHERE
                                        r.id_file = f.id_file)
        LIMIT 10000;
```

```
#2 - LEFT OUTER JOIN
```

```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=MEMORY
        SELECT SQL_NO_CACHE f.id_file
                FROM files f
                LEFT OUTER JOIN r_file_packer r0 ON f.id_file = r0.id_file
                LEFT OUTER JOIN r_file_archiver r1 ON f.id_file = r1.id_file
                WHERE
                        r0.id_file IS NULL AND
                        r1.id_file IS NULL
                LIMIT 100000;

DROP TEMPORARY TABLE struct_tmp_table;
```

```
#3 - DELETE FROM
```

```
CREATE TEMPORARY TABLE files_tmp_table ENGINE=MEMORY
        SELECT SQL_NO_CACHE f.id_file
                FROM files f
        LIMIT 160000;

DELETE tmp
        FROM files_tmp_table tmp
        INNER JOIN r_file_packer r0 ON tmp.id_file = r0.id_file
        WHERE
                tmp.id_file = r0.id_file;

DELETE tmp
        FROM files_tmp_table tmp
        INNER JOIN r_file_archiver r0 ON tmp.id_file = r0.id_file
        WHERE
                tmp.id_file = r0.id_file;

DROP TEMPORARY TABLE files_tmp_table;
```



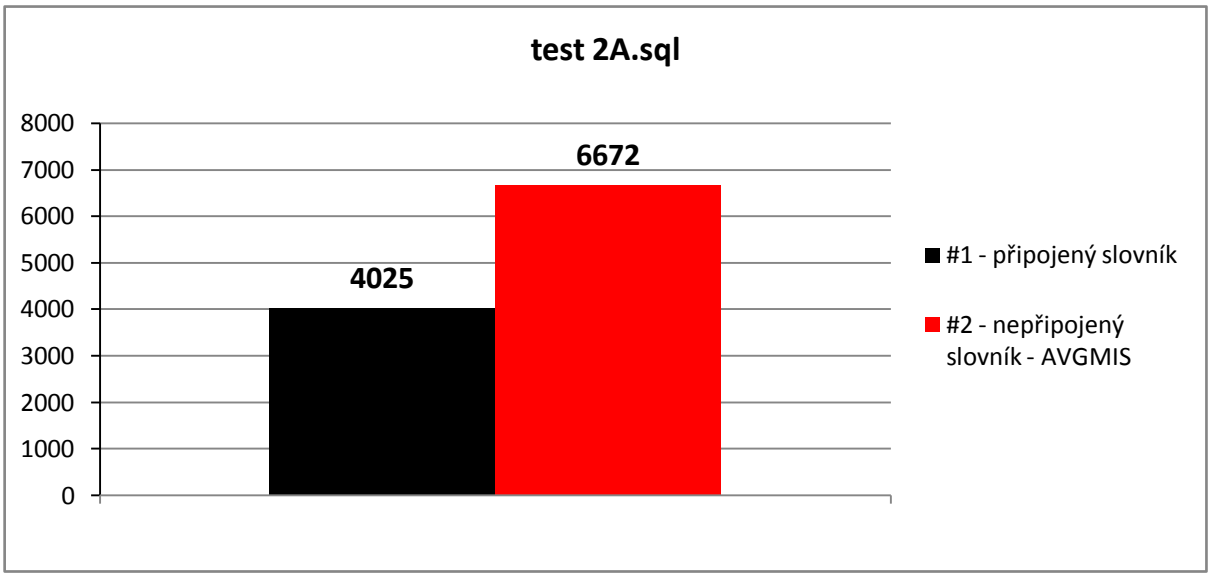
Test 2A.sql

```
#1 - připojený slovník
```

```
SELECT SQL_NO_CACHE f.id_file
        FROM files f
        INNER JOIN c_file_status c ON f.STATUS = f.status
        INNER JOIN r_files_iconhash r1 ON r1.id_file = f.id_file
        INNER JOIN c_iconhash c1 ON c1.id_iconhash = r1.id_iconhash
        WHERE
                c1.iconhash = 'd34adb97f75820a83d28491e3c333e34' AND
                c.meaning = 'infected'
        LIMIT 1000000;
```

```
#2 - nepřipojený slovník - AVGMIS
```

```
SELECT SQL_NO_CACHE f.id_file
        FROM files f
        INNER JOIN r_files_iconhash r1 ON r1.id_file = f.id_file
        INNER JOIN c_iconhash c1 ON c1.id_iconhash = r1.id_iconhash
        WHERE
                c1.iconhash='d34adb97f75820a83d28491e3c333e34' AND
                f.STATUS='2'
        LIMIT 1000000;
```



Test 2B.sql

```
#1 - Vícenásobný záporný packer s hashí z ikon – smíšený dotaz - vysoká četnost
```

```
SELECT SQL_NO_CACHE r1.id_file
        FROM r_files_iconhash r1
        INNER JOIN c_iconhash c1 ON r1.id_iconhash = c1.id_iconhash
        WHERE
                c1.iconhash = 'bb1648509f92f73d69726f1d2f1f2191' AND
                NOT EXISTS (
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_packer r
                                INNER JOIN c_packers c ON r.id_packer = c.id_packer
                                WHERE
                                        r.id_file = r1.id_file AND
                                        c.packer = 'UPX'
                ) AND
                NOT EXISTS (
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_packer r
                                INNER JOIN c_packers c ON r.id_packer = c.id_packer
                                WHERE
                                        r.id_file = r1.id_file AND
                                        c.packer = 'unknown'
                );
```

```
#2 - Vícenásobný záporný packer s hashí z ikon – smíšený dotaz - vysoká četnost- AVGMIS
```

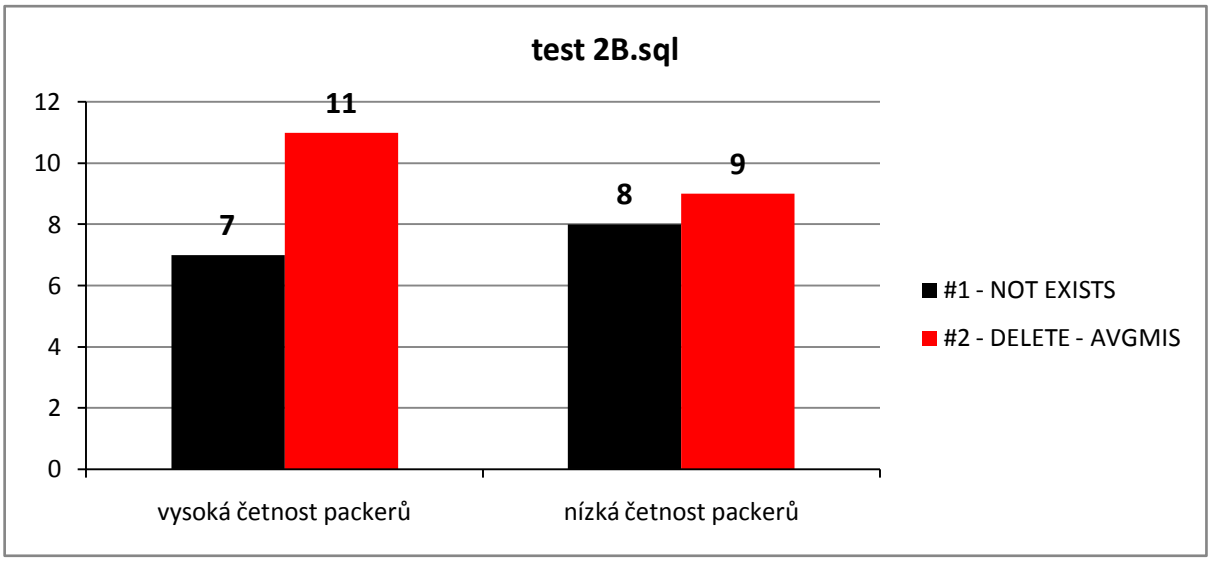
```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=Memory
        SELECT SQL_NO_CACHE r0.id_file
                FROM r_files_iconhash r0
                INNER JOIN c_iconhash c0 ON r0.id_iconhash = c0.id_iconhash
                WHERE
                        c0.iconhash='bb1648509f92f73d69726f1d2f1f2191';

DELETE tmp
        FROM struct_tmp_table tmp
        INNER JOIN r_file_packer r ON tmp.id_file = r.id_file
        INNER JOIN c_packers c ON r.id_packer = c.id_packer
        WHERE
                c.packer IN ('UPX','unknown');

DROP TEMPORARY TABLE struct_tmp_table;
```

```
#3 - Vícenásobný záporný packer s hashí z ikon – smíšený dotaz - nízká četnost
```

```
SELECT SQL_NO_CACHE r1.id_file
        FROM r_files_iconhash r1
        INNER JOIN c_iconhash c1 ON r1.id_iconhash = c1.id_iconhash
        WHERE
                c1.iconhash = 'bb1648509f92f73d69726f1d2f1f2191' AND
                NOT EXISTS (
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_packer r
                                INNER JOIN c_packers c ON r.id_packer = c.id_packer
                                WHERE
                                        r.id_file = r1.id_file AND
                                        c.packer = 'Yoda'
                ) AND
                NOT EXISTS (
                        SELECT SQL_NO_CACHE r.id_file
                                FROM r_file_packer r
                                INNER JOIN c_packers c ON r.id_packer = c.id_packer
                                WHERE
                                        r.id_file = r1.id_file AND
                                        c.packer = 'Molebox'
                );
```

#4 - Vícenásobný záporný packer s hashí z ikon – smíšený dotaz - nízká četnost- AVGMIS

```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=MyISAM
        SELECT SQL_NO_CACHE r0.id_file
                FROM r_files_iconhash r0
                INNER JOIN c_iconhash c0 ON r0.id_iconhash = c0.id_iconhash
                WHERE
                        c0.iconhash='bb1648509f92f73d69726f1d2f1f2191';

DELETE tmp
        FROM struct_tmp_table tmp
        INNER JOIN r_file_packer r ON tmp.id_file = r.id_file
        INNER JOIN c_packers c ON r.id_packer = c.id_packer
        WHERE
                c.packer IN ('Yoda','Molebox');

DROP TEMPORARY TABLE struct_tmp_table;
```



Test 2C.sql

#1 - Smíšený scénář s globálním záporem - vysoká četnost

```
SELECT SQL_NO_CACHE f.id_file
         FROM files f
         INNER JOIN r_files_certificates r1 ON f.id_file = r1.id_file
         INNER JOIN c_certificates c ON r1.id_certificate = c.id_certificate
         WHERE
                  c.sha256 = 'FEFE35C77B348940D2C1993F3C07DAA9611BA1EFBE983883F09C1B540340D4A4' AND
                  f.secthash = 'e0abd8a82571953721df063e050813d3' AND
                  NOT EXISTS(
                                    SELECT id_file
                                             FROM r_file_archiver r2
                                             WHERE r2.id_file = f.id_file
                  );
```

#2 - Smíšený scénář s globálním záporem - vysoká četnost - AVGMIS

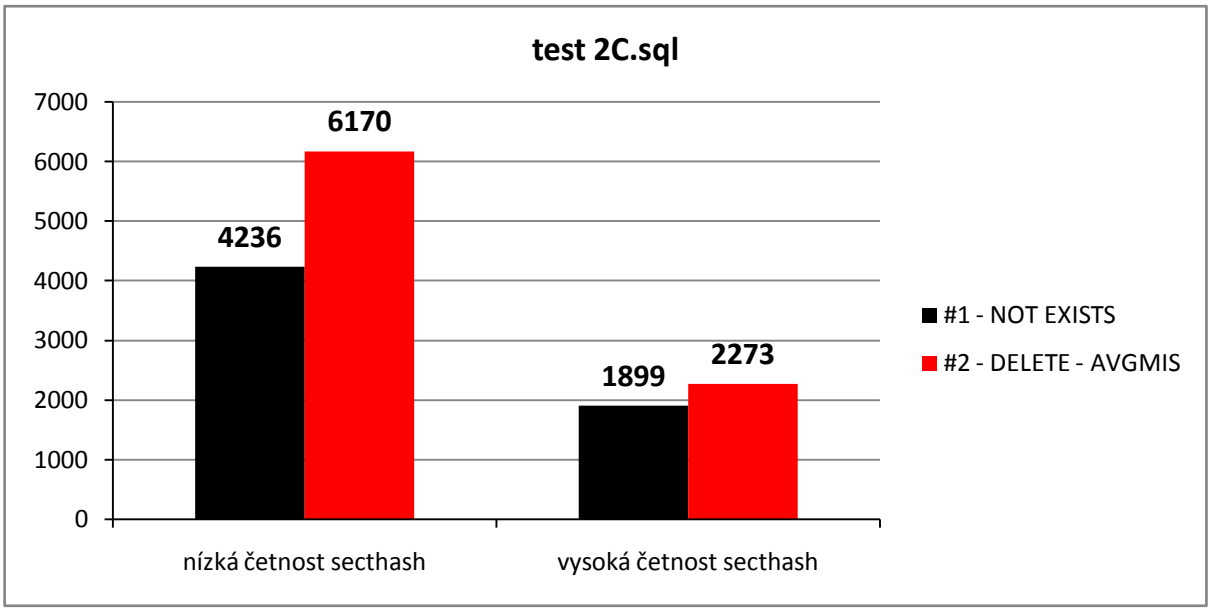
```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=Memory
         SELECT f.id_file
                  FROM files f
                  INNER JOIN r_files_certificates r0  ON f.id_file = r0.id_file
                  INNER JOIN c_certificates c0 ON r0.id_certificate = c0.id_certificate
                  WHERE
                           c0.SHA256 = 'FEFE35C77B348940D2C1993F3C07DAA9611BA1EFBE983883F09C1B540340D4A4' AND
                           f.SECTHASH='e0abd8a82571953721df063e050813d3';

         DELETE tmp
                  FROM struct_tmp_table tmp
                  INNER JOIN r_file_archiver r ON tmp.id_file = r.id_file
                  WHERE
                           tmp.id_file = r.id_file;

DROP TEMPORARY TABLE struct_tmp_table;
```

#3 - Smíšený scénář s globálním záporem - vysoká četnost

```
SELECT SQL_NO_CACHE f.id_file
         FROM files f
         INNER JOIN r_files_certificates r1 ON f.id_file = r1.id_file
         INNER JOIN c_certificates c ON r1.id_certificate = c.id_certificate
         WHERE
                  c.sha256 = 'FEFE35C77B348940D2C1993F3C07DAA9611BA1EFBE983883F09C1B540340D4A4' AND
                  f.secthash = '67d7f0c7a3b2c4175269aaa3b7e225a6' AND
                  NOT EXISTS(
                                    SELECT id_file
                                             FROM r_file_archiver r2
                                             WHERE r2.id_file = f.id_file
                  );
```

#4 - Smíšený scénář s globálním záporem - vysoká četnost - AVGMIS

```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=Memory
         SELECT f.id_file
                  FROM files f
                  INNER JOIN r_files_certificates r0  ON f.id_file = r0.id_file
                  INNER JOIN c_certificates c0 ON r0.id_certificate = c0.id_certificate
                  WHERE
                           c0.SHA256='FEFE35C77B348940D2C1993F3C07DAA9611BA1EFBE983883F09C1B540340D4A4' AND
                           f.SECTHASH='67d7f0c7a3b2c4175269aaa3b7e225a6';

         DELETE tmp
                  FROM struct_tmp_table tmp
                  INNER JOIN r_file_archiver r ON tmp.id_file = r.id_file
                  WHERE
                           tmp.id_file = r.id_file;

DROP TEMPORARY TABLE struct_tmp_table;
```


test 2C.sql
7000
6000
5000
4000
3000
2000
1000
0
4236
6170
1899
2273
nízká četnost secthash
vysoká četnost secthash
#1 - NOT EXISTS
#2 - DELETE - AVGMIS

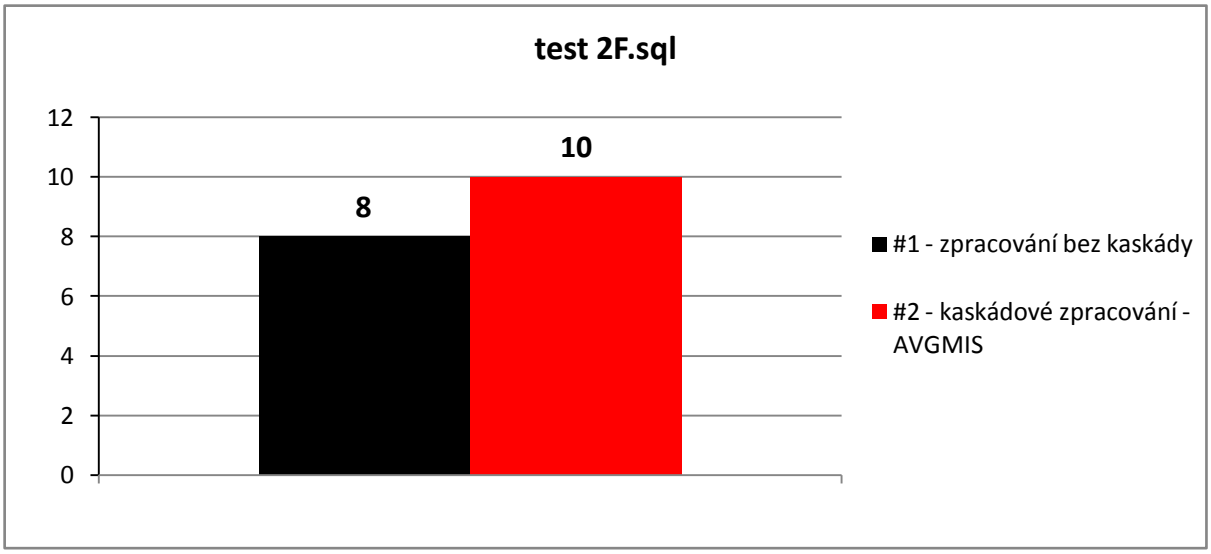
Test 2F.sql

```
#1 - Komplexní požadavek – více stupňů kaskády
```

```
SELECT SQL_NO_CACHE f.id_file
        FROM files f
        INNER JOIN r_file_packer r0 ON r0.id_file = f.id_file
        INNER JOIN c_packers c0 ON r0.id_packer = c0.id_packer
        INNER JOIN r_file_archiver r1 ON r1.id_file = f.id_file
        INNER JOIN c_archivers c2 ON r1.id_archiver = c2.id_archiver
        INNER JOIN c_file_status c1 ON f.status = c1.status
        WHERE
                f.VERINFO_CompanyName LIKE 'Microsoft%' AND
                c1.meaning = 'infected' AND
                c0.packer = 'UPX' AND
                c2.archiver IN ('NSIS','CAB','ZIP')
        LIMIT 100;
```

```
#2 - Komplexní požadavek – více stupňů kaskády - AVGMIS
```

```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=Memory
        SELECT r0.id_file
                FROM r_file_packer r0
                INNER JOIN c_packers c0 ON r0.id_packer = c0.id_packer
                INNER JOIN r_file_archiver r2 ON r2.id_file = r0.id_file
                INNER JOIN c_archivers c2 ON c2.id_archiver = r2.id_archiver
                WHERE
                        c0.packer = 'UPX' AND
                        c2.archiver IN ('NSIS','CAB','ZIP')
        LIMIT 50000;

CREATE TEMPORARY TABLE files_tmp_table ENGINE=Memory
        SELECT tmp.id_file
                FROM struct_tmp_table tmp
                INNER JOIN files f ON tmp.id_file = f.id_file
                WHERE
                        f.STATUS = 2 AND
                        f.VERINFO_CompanyName LIKE 'Microsoft%'
        LIMIT 100;

DROP TEMPORARY TABLE files_tmp_table;
DROP TEMPORARY TABLE struct_tmp_table;
```



Test 2G.sql

```
#1 - Úplnost či rychlost - rychlost - vysoká četnost packerů
```

```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=MyISAM
        SELECT r0.id_file
                FROM r_file_packer r0
                INNER JOIN c_packers c0 ON r0.id_packer = c0.id_packer
                WHERE
                        c0.packer = 'UPX'
        LIMIT 50000;

        DELETE tmp
                FROM struct_tmp_table tmp
                INNER JOIN r_file_archiver r ON tmp.id_file = r.id_file
                WHERE tmp.id_file = r.id_file;

CREATE TEMPORARY TABLE files_tmp_table ENGINE=MyISAM
        SELECT tmp.id_file
                FROM struct_tmp_table tmp
                INNER JOIN files f ON tmp.id_file = f.id_file
                WHERE
                        f.VERINFO_CompanyName LIKE 'Micr%' AND
                        f.STATUS = 2
        LIMIT 30;

DROP TEMPORARY TABLE files_tmp_table;
DROP TEMPORARY TABLE struct_tmp_table;
```

```
#2 - Úplnost či rychlost - úplnost - vysoká četnost packerů - AVGMIS
```

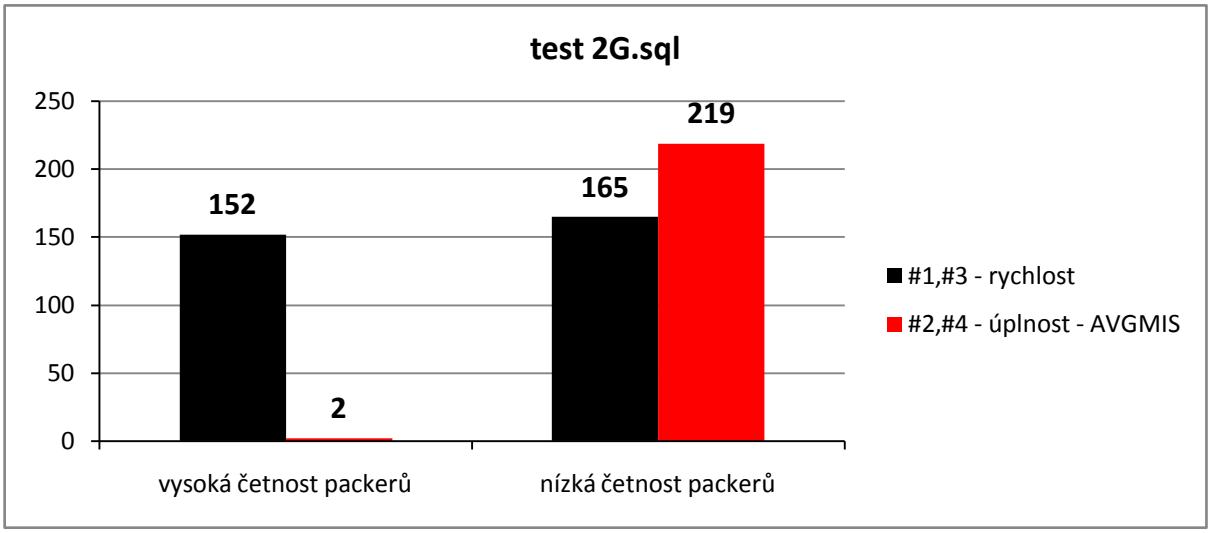
```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=MyISAM
        SELECT r0.id_file
                FROM r_file_packer r0
                INNER JOIN c_packers c0 ON r0.id_packer = c0.id_packer
                WHERE
                        c0.packer = 'UPX';

        DELETE tmp
                FROM struct_tmp_table tmp
                INNER JOIN r_file_archiver r ON tmp.id_file = r.id_file
                WHERE tmp.id_file = r.id_file;

CREATE TEMPORARY TABLE files_tmp_table ENGINE=MyISAM
        SELECT tmp.id_file
                FROM struct_tmp_table tmp
                INNER JOIN files f ON tmp.id_file = f.id_file
                WHERE
                        f.VERINFO_CompanyName LIKE 'Micr%' AND
                        f.STATUS = 2
        LIMIT 30;

DROP TEMPORARY TABLE files_tmp_table;
DROP TEMPORARY TABLE struct_tmp_table;
```

```
#3 - Úplnost či rychlost -rychlost - nízká četnost packerů
```

```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=MyISAM
        SELECT r0.id_file
                FROM r_file_packer r0
                INNER JOIN c_packers c0 ON r0.id_packer = c0.id_packer
                INNER JOIN files r1 ON r1.id_file = r0.id_file
                WHERE
                        c0.packer = 'Yoda'
        LIMIT 50000;

        DELETE tmp
                FROM struct_tmp_table tmp
                INNER JOIN r_file_archiver r ON tmp.id_file = r.id_file
                WHERE tmp.id_file = r.id_file;

CREATE TEMPORARY TABLE files_tmp_table ENGINE=MyISAM
        SELECT tmp.id_file
                FROM struct_tmp_table tmp
                INNER JOIN files f ON tmp.id_file = f.id_file
                WHERE
                        f.VERINFO_CompanyName LIKE 'Micr%' AND
                        f.STATUS = 2
        LIMIT 30;

DROP TEMPORARY TABLE files_tmp_table;
```

#4 - Úplnost či rychlost - úplnost - nízká četnost packerů - AVGMIS

```
CREATE TEMPORARY TABLE struct_tmp_table ENGINE=MyISAM
        SELECT r0.id_file
                FROM r_file_packer r0
                INNER JOIN c_packers c0 ON r0.id_packer = c0.id_packer
                INNER JOIN files r1 ON r1.id_file = r0.id_file
                WHERE
                        c0.packer = 'Yoda';

        DELETE tmp
                FROM struct_tmp_table tmp
                INNER JOIN r_file_archiver r ON tmp.id_file = r.id_file
                WHERE tmp.id_file = r.id_file;

CREATE TEMPORARY TABLE files_tmp_table ENGINE=MyISAM
        SELECT tmp.id_file
                FROM struct_tmp_table tmp
                INNER JOIN files f ON tmp.id_file = f.id_file
                WHERE
                        f.VERINFO_CompanyName LIKE 'Micr%' AND
                        f.STATUS = 2
        LIMIT 30;

DROP TEMPORARY TABLE files_tmp_table;
```



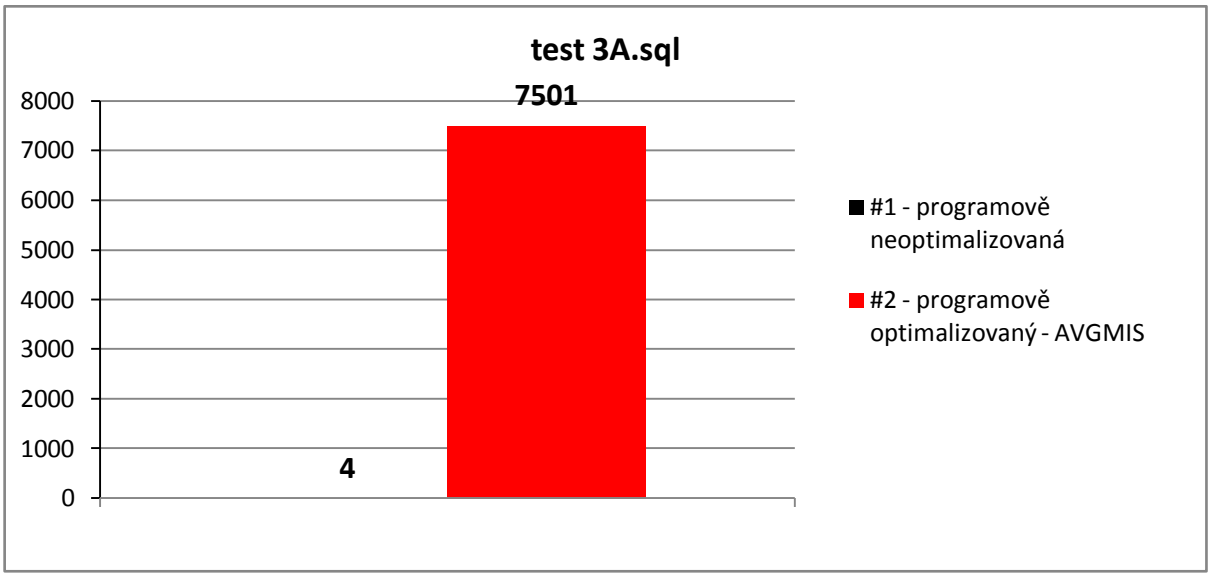
Test 3A.sql

```
#1 - Vnitřní optimalizace detekčních skupin - programově neoptimlizováno
```

```
(SELECT r0.id_file as col
    FROM r_files_det_avg r0
    INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
    WHERE
        c0.det_avg LIKE 'Adware%') UNION
(SELECT r0.id_file as col
    FROM r_files_det_avg r0
    INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
    WHERE
        c0.det_avg LIKE 'Adware%' AND
        NOT EXISTS(
            SELECT *
                FROM r_files_det_kav r2
                WHERE
                    r2.id_file = r0.id_file))
LIMIT 30;
```

```
#2 - Vnitřní optimalizace detekčních skupin - programově optimlizováno - AVGMIS
```

```
SELECT r0.id_file
    FROM r_files_det_avg r0
    INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
    WHERE
                c0.det_avg LIKE 'Adware%'
    LIMIT 30;
```



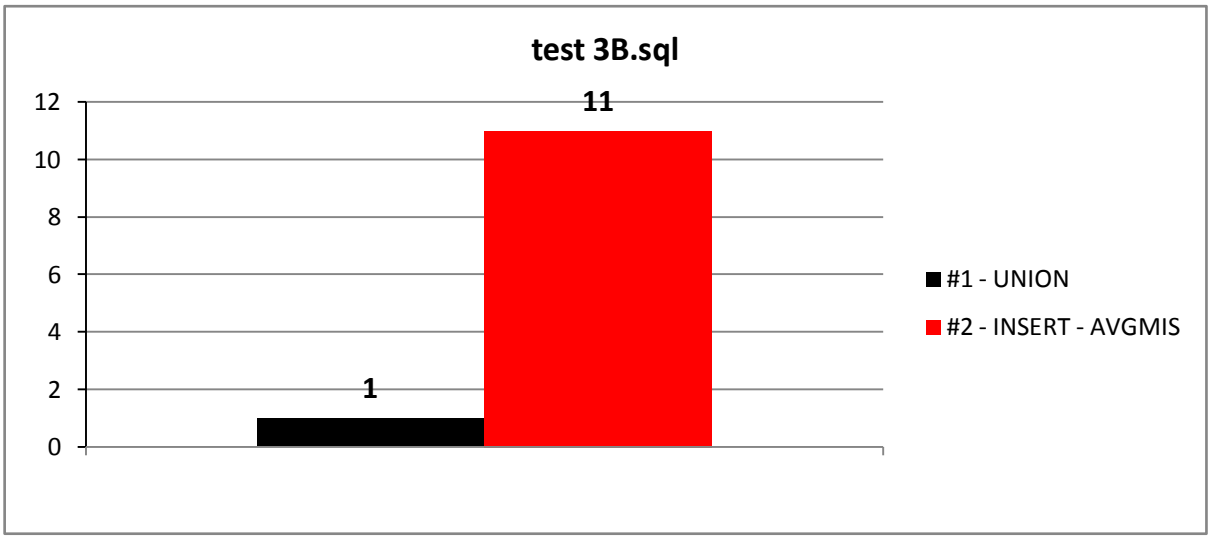
Test 3B.sql

#1 - Vícenásobné detekční skupiny - UNION

```
(SELECT r0.id_file as col
    FROM r_files_det_avg r0
    INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
    WHERE
        c0.det_avg LIKE '%Agent.6.Au%' AND
        NOT EXISTS(
            SELECT *
                FROM r_files_det_mcafee r2
                WHERE
                    r2.id_file = r0.id_file))
UNION
(SELECT r0.id_file as col
    FROM r_files_det_avg r0
    INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
    WHERE
    c0.det_avg LIKE '%VBS/Downloader.agent%' AND
    NOT EXISTS(
            SELECT *
                FROM r_files_det_mcafee r2
                WHERE
                    r2.id_file = r0.id_file
    )
)
UNION
(SELECT r0.id_file as col
    FROM r_files_det_avg r0
    INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
    WHERE
    c0.det_avg LIKE 'Trojan Horse%' AND
    NOT EXISTS(
            SELECT *
                FROM r_files_det_avira r2
                WHERE
                    r2.id_file = r0.id_file
    )
)
LIMIT 30;
```

#2 - Vícenásobné detekční skupiny - INSERT - AVGMIS

```
CREATE TEMPORARY TABLE detection_tmp_table ( id_file INT NOT NULL PRIMARY KEY ) ENGINE=MyISAM;

INSERT IGNORE INTO detection_tmp_table
         SELECT r0.id_file
                  FROM r_files_det_avg r0
                  INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
                  LEFT JOIN r_files_det_mcafee r1 ON r0.id_file = r1.id_file
                  WHERE
                           c0.det_avg LIKE '%Agent.6.Au%' AND
                           r1.id_file IS NULL
                  LIMIT 30;

INSERT IGNORE INTO detection_tmp_table
         SELECT r0.id_file
                  FROM r_files_det_avg r0
                  INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
                  LEFT JOIN r_files_det_mcafee r1 ON r0.id_file = r1.id_file
                  WHERE
                           c0.det_avg LIKE '%VBS/Downloader.agent%' AND
                           r1.id_file IS NULL
                  LIMIT 30;

INSERT IGNORE INTO detection_tmp_table
         SELECT r0.id_file
                  FROM r_files_det_avg r0
                  INNER JOIN c_det_avg c0 ON r0.id_det_avg = c0.id_det_avg
                  LEFT JOIN r_files_det_avira r1 ON r0.id_file = r1.id_file
                  WHERE
                           c0.det_avg LIKE 'Trojan Horse%' AND
                           r1.id_file IS NULL
                  LIMIT 30;

DROP TEMPORARY TABLE detection_tmp_table;
```


test 3B.sql
12
10
8
6
4
2
0
11
1
#1 - UNION
#2 - INSERT - AVGMIS

14 Příloha D

Tato příloha se zabývá obsahem datového nosiče, který je přiložen k diplomové práci. V kořenovém adresáři datového nosiče jsou tři složky: `docs`, `sources` a `tests`. První složka `docs` obsahuje elektronickou kopii diplomové práce a dále pak různé další podklady, které byly vytvořeny během fáze návrhu aplikace (diagramy řešící zpracování jednotlivých scénářů a skupin – umístěné ve složce `flow`, návrh kaskádového zpracování a řízení priorit zpracování – umístěny ve složce `search_strategy`, návrh a implementace záznamů jednotlivých hledání – umístěny ve složce `view_log_avgmis`).

Složka `sources` obsahuje kompletní řešení Visual Studia 2010 k aplikaci AVGMIS (uvnitř složky `AVGMIS_VS`) a dále pak řešení Visual Studia 2010 pro program `QueryCounter` (složka `QueryCounter`). Poslední složkou ve složce `sources` je stejnojmenná složka, která obsahuje zdrojové kódy aplikační logiky, která vytváří aplikační jádro aplikace AVGMIS.

Složka `tests` obsahuje původní testovací skripty, jejichž obsah je uveden zde v diplomové práci, konkrétně v kapitole o testování a dále v příloze C. Současně jsou zde obsaženy i výsledky testů (soubor `result.xlsx`).